Java certification success, Part 4: SCEA

Skill Level: Intermediate

Sivasundaram Umapathy (authors@whizlabs.com)
Programmer

20 May 2005

This tutorial aims to help SCEA certification aspirants clear the first part of the SCEA certification exam, a knowledge-based, multiple-choice exam. The tutorial introduces the reader to the concepts and then builds upon them to cover other topics such as common architectures, legacy connectivity, Enterprise JavaBeans technology, the Enterprise JavaBeans container model, protocols, applicability of J2EE technology, design patterns, messaging, internationalization, and security. Readers' understanding is then reinforced through examples and practice questions and guides them to various useful resources for SCEA certification exam preparation.

Section 1. Getting started

Preparing for SCEA

The Sun Certified Enterprise Architect (SCEA) exam is for professionals who design and build enterprise solutions using Java™ 2 Platform, Enterprise Edition (J2EE) technology in a robust, scalable, secure, and flexible way. This exam is the highest title in the certification path for the J2EE track. Although this exam logically follows the Programmer and Developer exams, there are no prerequisites for taking this exam. Hands-on experience designing J2EE solutions will be helpful in clearing this exam at the first go. This three-part exam comprises a knowledge-based, multiple-choice exam, an assignment, and an essay exam.

What's in this tutorial?

This tutorial focuses on the Part 1 knowledge-based, multiple-choice exam. This exam differs from the Programmer and Developer exams because it tests a wide range of topics that could be aptly summarized by the phrase "mile wide and inch deep."

This tutorial covers the following main sections.

- **Section 1: Concepts**
- **Section 2: Common architectures**
- **Section 3: Legacy connectivity**
- **Section 4: Enterprise JavaBeans**
- **Section 5: Enterprise JavaBeans container model**
- **Section 6: Protocols**
- **Section 7: Applicability of J2EE**
- **Section 8: Design patterns**
- **Section 9: Messaging**
- **Section 10: Internationalization**
- **Section 11: Security**

Although the exam does not focus on a particular J2EE version, the questions relate to J2EE version 1.2. So, you might not find the latest concepts, such as message-driven beans, Web services, and other features of later J2EE versions.

Each section of this tutorial deals with a single objective. Wherever required, appropriate diagrams and examples have been provided to ease understanding of the subjects. This tutorial cannot and should not be used as the only source of reading as it does not elaborate much on each topic; rather, it helps you to prepare for the exam by concentrating on the key points tested in the exam.

Each chapter ends with a summary and mock questions that represent the actual exam pattern. These are not real questions from the exam, but they help you to understand the extent to which the objectives are tested in the exam. Explanations about the correct and incorrect choices are included to give you a better understanding of the concepts.

Section 2. Concepts

Introduction

Software modeling involves designing software applications before coding. Creating a model helps you to understand the system better -- before it is developed. Unified Modeling Language (UML) is one such modeling language you can use to specify,

visualize, and document models of software systems, including their structure and design, in a way that meets all of your requirements. It is important to remember that UML is not a process methodology; it's just a modeling language. In practice, UML is often used with a process methodology. The current exam is based on UML version 1.x.

UML has three major elements: building blocks, relationship rules, and common mechanisms. Let's examine them one by one.

UML: Building blocks

UML building blocks can be divided into three categories:

- Elements
- Relationships
- Diagrams

Elements are abstractions; relationships tie these elements together; and diagrams group the collection of related elements by means of relationships.

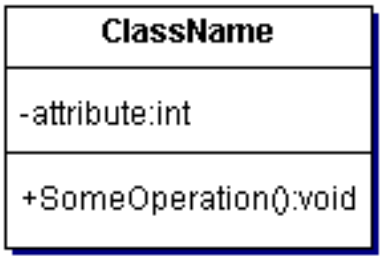
UML: Elements

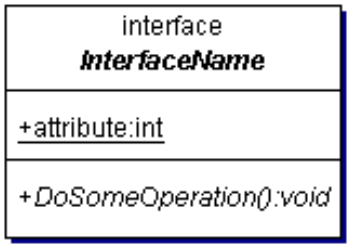
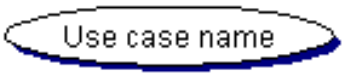
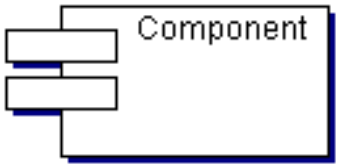
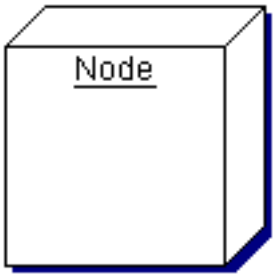
There are four types of elements:

- Structural
- Behavioral
- Grouping
- Annotational

Structural

These elements are similar to the nouns of a language.

Name	Definition	Notation
Class	Set of objects that share the same attributes, operations, relationships, and semantics. It is represented by a rectangle containing three areas: name of the class, attributes (properties) of the class, and the operations (methods) of the class.	
Interface	A collection of operations that specify a service of a class or component. This is also	

	represented by a rectangle with three areas representing the name of the interface, attributes, and the operations of the interface. This has an addition of the word "interface" above the interface name.	
Collaboration	Defines an interaction and is a combination of roles and other elements that work together to provide some cooperative behavior bigger than the sum of all the elements. This is represented by a dashed-line ellipse.	
Use case	A description of a set of actions the system performs to yield an observable result of value to an actor. This is represented by an ellipse with the use case name inside the ellipse.	
Active class	A class whose instances are active objects. They own one or more processes or threads to initiate control activity.	
Component	A physical and replaceable part of the system that conforms to and provides the realization of a set of interfaces.	
Node	A physical element that exists at runtime and represents a computational resource having some memory and processing capability.	

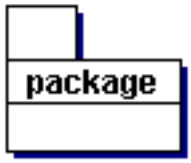
Behavioral

Defines the dynamic part of the UML elements.

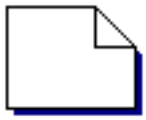
Name	Definition	Notation

Interaction	Comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose.	
Statemachine	Specifies the sequence of states an object or interaction goes through during its lifetime in response to events, together with its response to those events.	

Grouping

Name	Definition	Notation
Package	A general-purpose mechanism for organizing elements into groups.	

Annotational

Name	Definition	Notation
Note	A symbol for rendering comments you want attached to other elements or collections of elements.	

UML: Relationships

Relationships

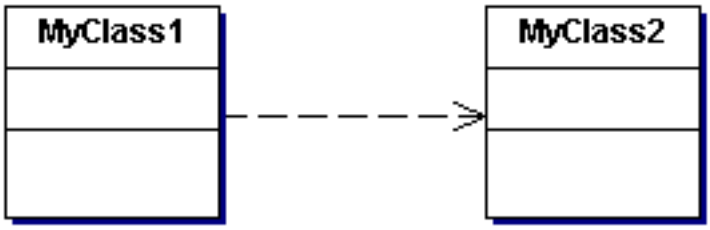
There are four types of UML relationships:

- Dependency
- Association
- Generalization
- Realization

Dependency

A dependency is a semantic relationship between two elements in which a change in one element can affect the semantics of another element. The arrow indicates the

direction of dependency. In the following diagram, MyClass1 has a dependency relationship with MyClass2. The change in MyClass2 affects MyClass1.

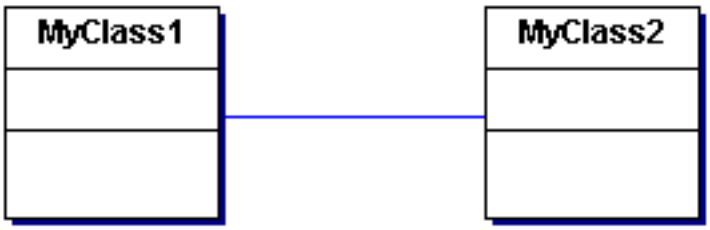


Association

An association is a structural relationship that describes a set of connections between objects. An association may have a multiplicity at each end that represents the number of elements the element at the other end of the association has with the end that specifies the multiplicity.

Multiplicity	Meaning
1	One and only one
0..* or *	Zero, one, or many
1..*	One or many
a..b	Between a and b
a,b	a or b

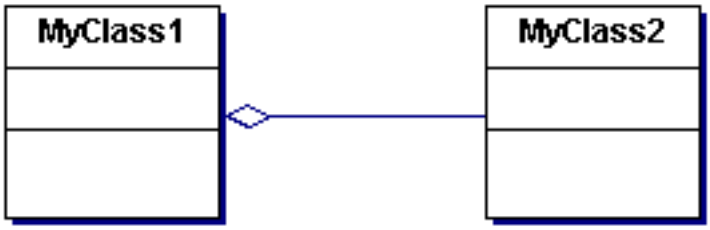
An association may be directed  or undirected . If it is undirected, then it has not been decided if the association is directed, or the association is bi-directional. In the following diagram, the association is undirected.



There are two special types of associations:

- Aggregation
- Composition

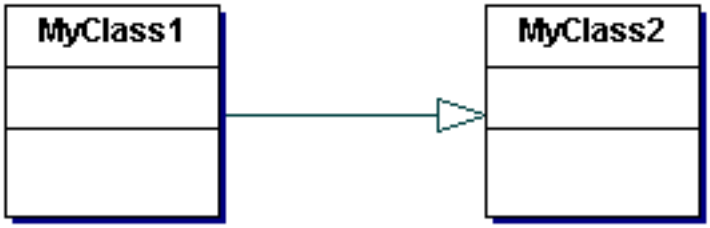
Aggregation represents the relationship between the whole and the part. One end of the association is designated the aggregate while the other end is unmarked. In this diagram, Myclass2 is a part of Myclass1.



Composition also represents the whole and the part relationship, but it is a stronger form of aggregation. There is an additional constraint that an object might be part of only one composite and that the composite object has the responsibility for the lifetime of all its parts -- that is, for their creation and destruction. In the following diagram, MyClass1 cannot exist without MyClass2.

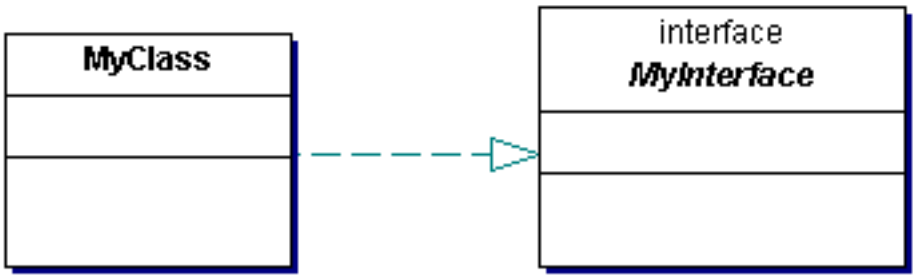
Generalization

Generalization is a parent-child relationship. MyClass2 is the super class, and MyClass1 is the subclass. With Java programming language, you implement generalization by subclassing using the `extends` keyword.



Realization

Realization is the relationship between the interface and the implemented class. In Java language, you implement realization by implementing an interface using the `implements` keyword.



Common mechanisms

Let's now discuss the following common mechanisms:

- Specifications
- Adornments
- Common divisions
- Extensibility mechanisms

Specifications

A specification is textual statements representing the syntax and semantics of the building blocks. A class can specify all, or only part of, the attributes, operations, and behaviors.

Adornments

You use adornments to represent additional state, for example, whether a class is abstract, or the visibility of attributes or operations (+public, #protected, -private).

Common divisions

If the element name is underlined in the class diagram, it an instance of a class (class is an instance of class). :class represents an anonymous instance of class, and named:class is a named instance of class.

Extensibility mechanisms

Extensibility mechanisms allow you to customize and expand UML. Stereotypes, tagged values, and constraints are some of the mechanisms.

UML: Diagrams

We will discuss the following UML diagrams:

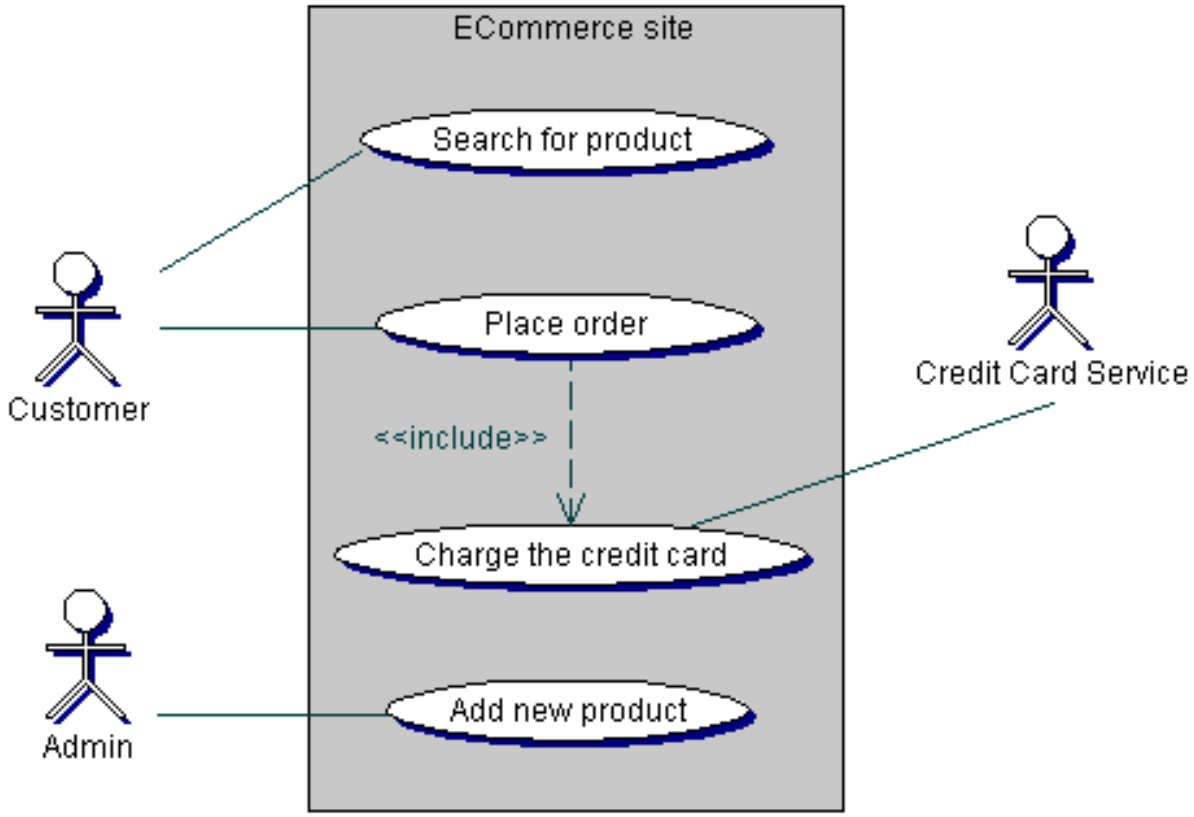
- Use-case diagram
- Class diagram
- Package diagram
- Interaction diagram
- Statechart diagram
- Activity diagram
- Component diagram
- Deployment diagram

Use-case diagram

A diagram shows the relationships among actors and use cases within a system. It helps the system analyst to elaborate the requirements from an end-user view. It is drawn as a graph of actors, a set of use cases enclosed by a system boundary (a rectangle), associations between the actors and the use cases, and generalization among the actors.

A use case is a summary of scenarios for a single task or goal. An actor is a person or a system that initiates the events involved in that task. In the following use-case diagram, there are three actors: customer, admin, and the credit card service. The ellipses represent the use case (business process), and the actors access the different use cases. The connection between actor and use case is called communication. The "charge the credit card" use case is factored out as a separate use case and modeled using an include relationship. This means that the other use cases that need to charge the customer can reuse the common factored out "charge the credit card" use case. The system boundary separates the system from the actors and is represented by the ECommerce site rectangle.

Other possible relationships in a use-case diagram are the extend relationship, which indicates that one use case is a variation of another, and generalization that is used to represent inheritance among use cases.



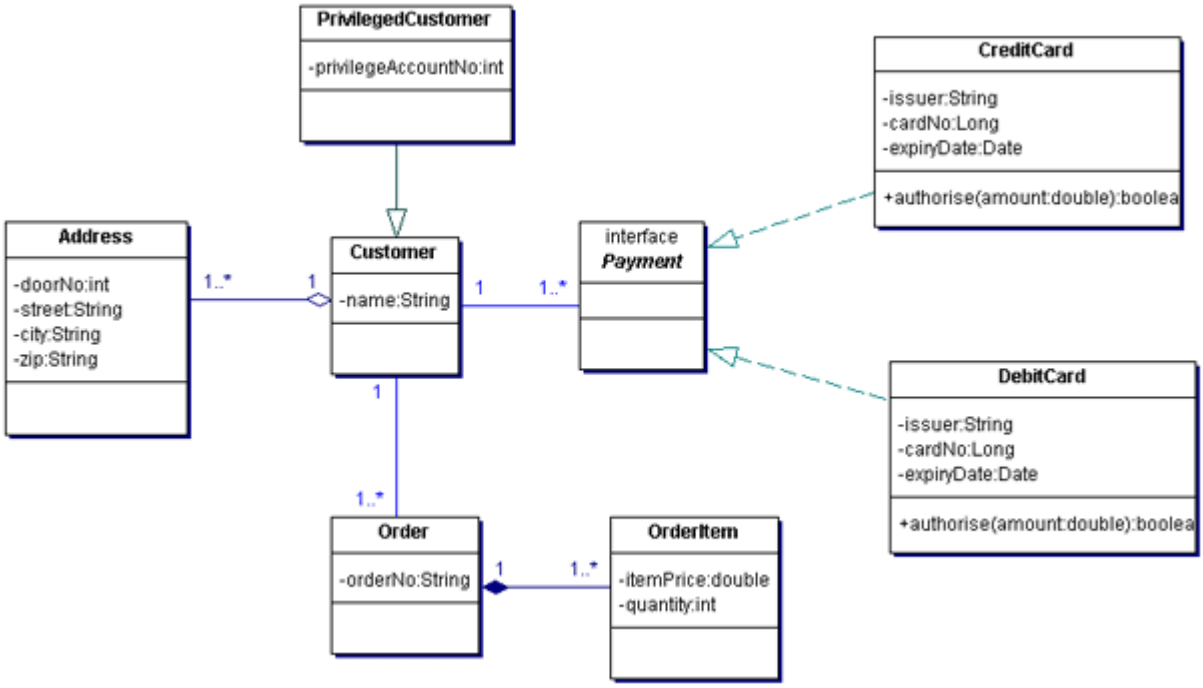
Class diagram

A class diagram describes the static structure of the symbols in the system. It is a graphic presentation of the static view that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships. Classes are arranged in hierarchies sharing common structure and behavior, and

are associated with other classes. A class diagram might contain certain behavioral elements, such as operations, but their dynamics are usually expressed in other diagrams, such as statechart diagrams and collaboration diagrams.

A single class diagram might not be enough to show the entire static view. So, generally you might find multiple class diagrams based on logical boundaries such as packages.

In the following class diagram, the static structure is defined around the main entity Customer that is connected to various other classes, such as Address, Order, and Payment interface. A customer can have a collection of Addresses (shipping address, billing address, alternate shipping address, and so on) modeled by the aggregation. The customer also has an association relation with the Payment interface and the Order class. The Payment interface can either be a CreditCard or a DebitCard, which are two specific realization models of the Payment interface. Each order has many OrderItems attached to it. Because the OrderItem cannot live without the Order, the relationship is modeled as a composition. The PrivilegedCustomer is a special form of Customer who gets loyalty points for the purchases made and is extended from the Customer using a generalization relation. The navigability shows the direction in which an association can be traversed. The multiplicity indicates the possible instances.

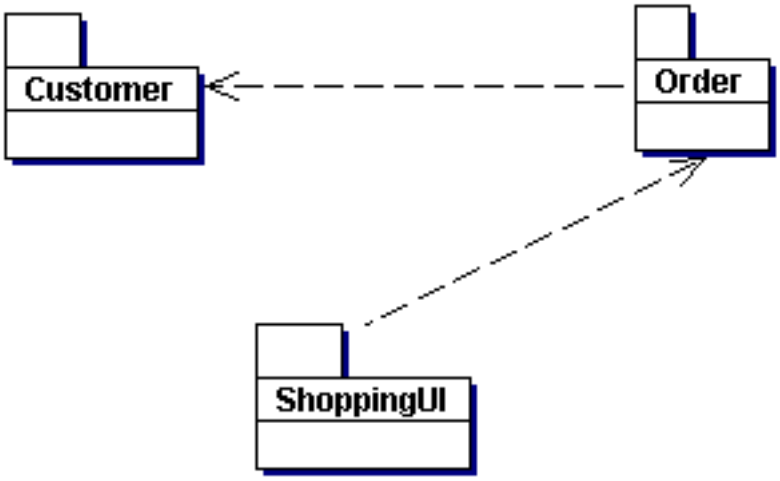


UML: Diagrams, continued

Package diagram

A package diagram shows the organization of systems in groups and can be considered as a special kind of class diagram. The classes are grouped into packages and represented by rectangles with a tab on the extreme right. The dotted

arrows represent the dependencies. In the following example, there are three packages: Customer, Order, and ShoppingUI. Changes in the Customer package's classes will affect the Order package's classes, and they are therefore represented by a dependency relation.



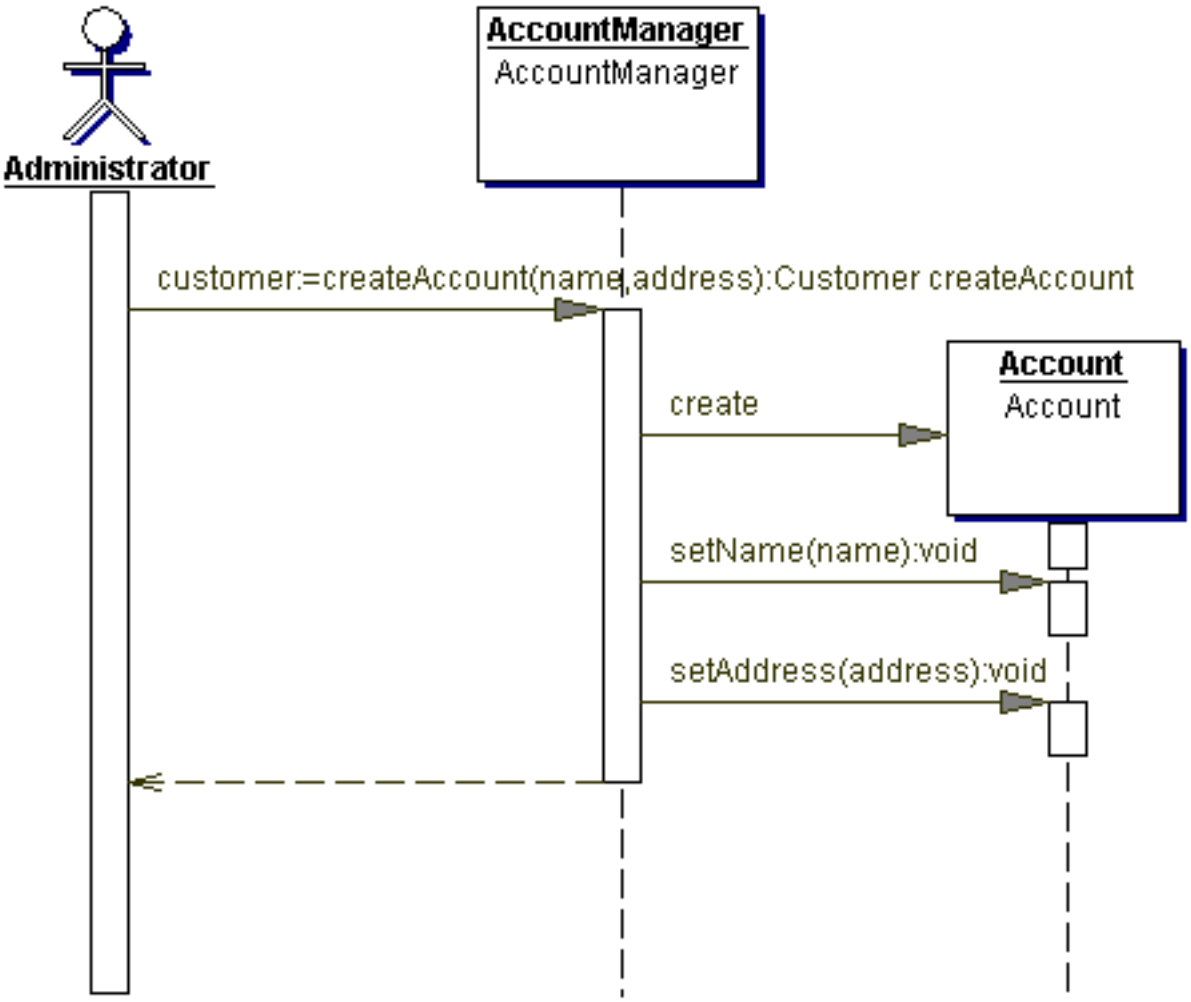
Interaction diagram

As the name implies, the interaction diagram emphasizes object interactions, consisting of a set of objects and their relationships, along with the messages exchanged between them. Both sequence and collaboration diagrams are forms of interaction diagrams. Although they use the same underlying information, each of them represents a specific view. It is worth noting that these diagrams are isomorphic, meaning that from one diagram you can derive the other.

A **sequence** diagram shows an interaction arranged in time-order sequence. So you can see the objects by their lifelines and the messages they exchange, arranged in time sequence.

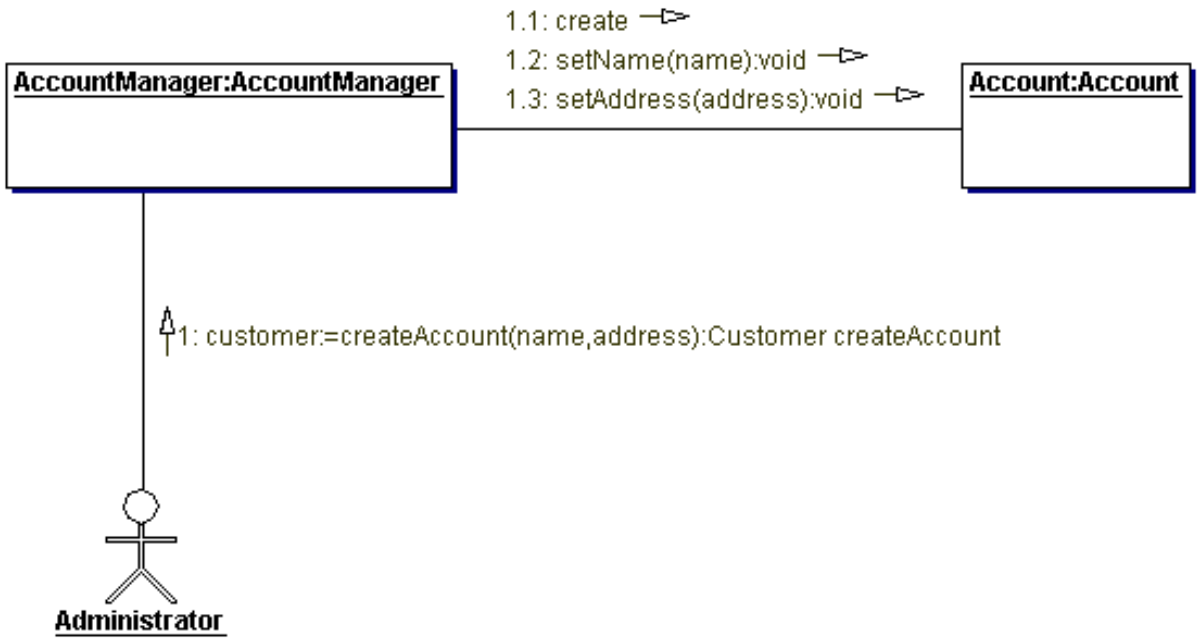
In the following sequence, the Administrator initiates the sequence by sending the `createAccount(name,address)` message to the AccountManager class to create a new customer account. The AccountManager now creates a new Account class and sets the various properties of the class by sending messages, such as `setName(name)` and `setAddress(address)`. Finally, the newly created Account object is returned.

The messages are represented by arrows, and the vertical line is the lifeline of the object, indicating when the object is created and how long it exists.



A **collaboration** diagram shows an interaction arranged around the objects that perform the operations. Hence, it focuses more on structural organization of the objects that send and receive messages.

In the following collaboration diagram, the rectangles represent the objects, and objects are labeled using the object name followed by the class name separated by a colon (:). The messages have a sequence number, and the initiating first message starts with the number 1. The messages, such as `create` and `setName()`, have the same decimal prefix to indicate that they are sent as part of the same call; their increasing suffixes indicate the order in which they occur.

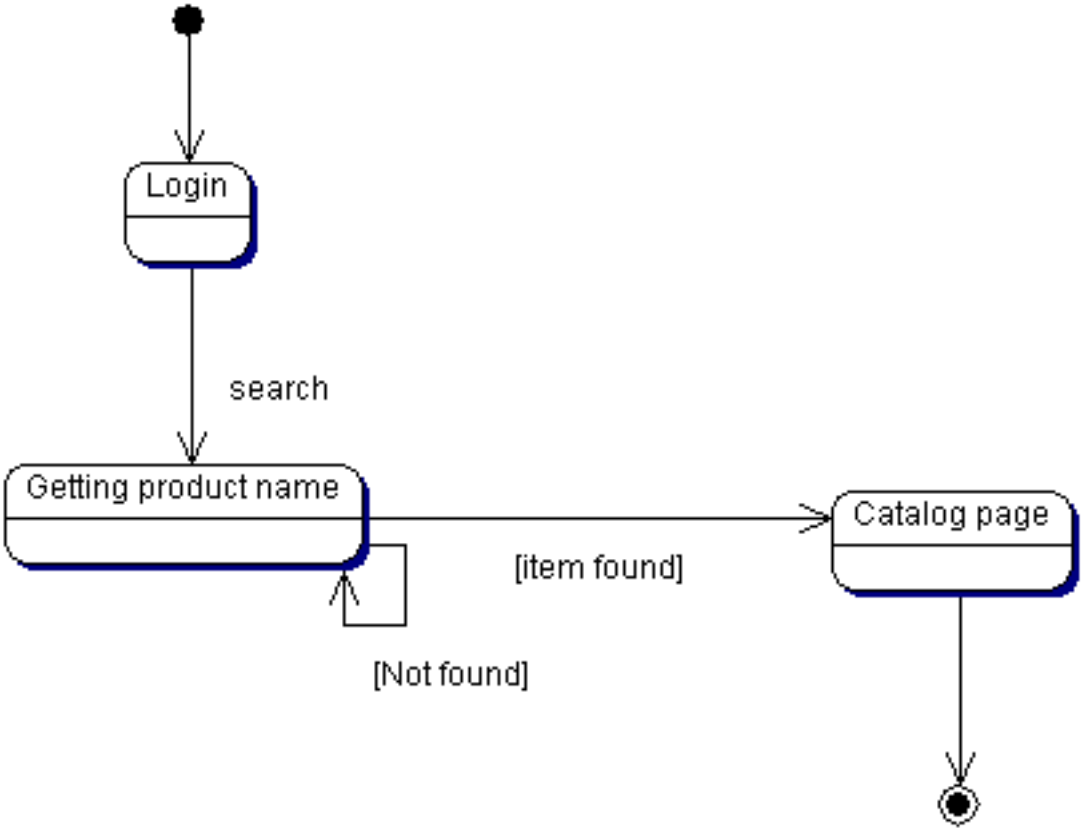


UML: Diagrams, continued

Statechart diagram

A statechart diagram shows a state machine consisting of states, transitions, events, and activities. It provides a detailed picture of how a specific symbol changes states. A state refers to the value associated with a specific attribute of an object and to any actions or side effects that occur when the attribute's value changes.

In the following example, there are three states: Login, Getting product name, and Catalog page, represented by rounded rectangles. Arrows represent the transitions. The initial state represented by the black circle is a dummy state to start the action. The events or conditions that trigger the transaction are written as labels on the arrows. After the Login state, the application moves to the Getting product name state. If the product is found (represented by the [item found] condition), the application transitions to the Catalog page. On the other hand, if the product is not found, the self-transition to the same state happens. The final state is represented by a concentric white/black circle.

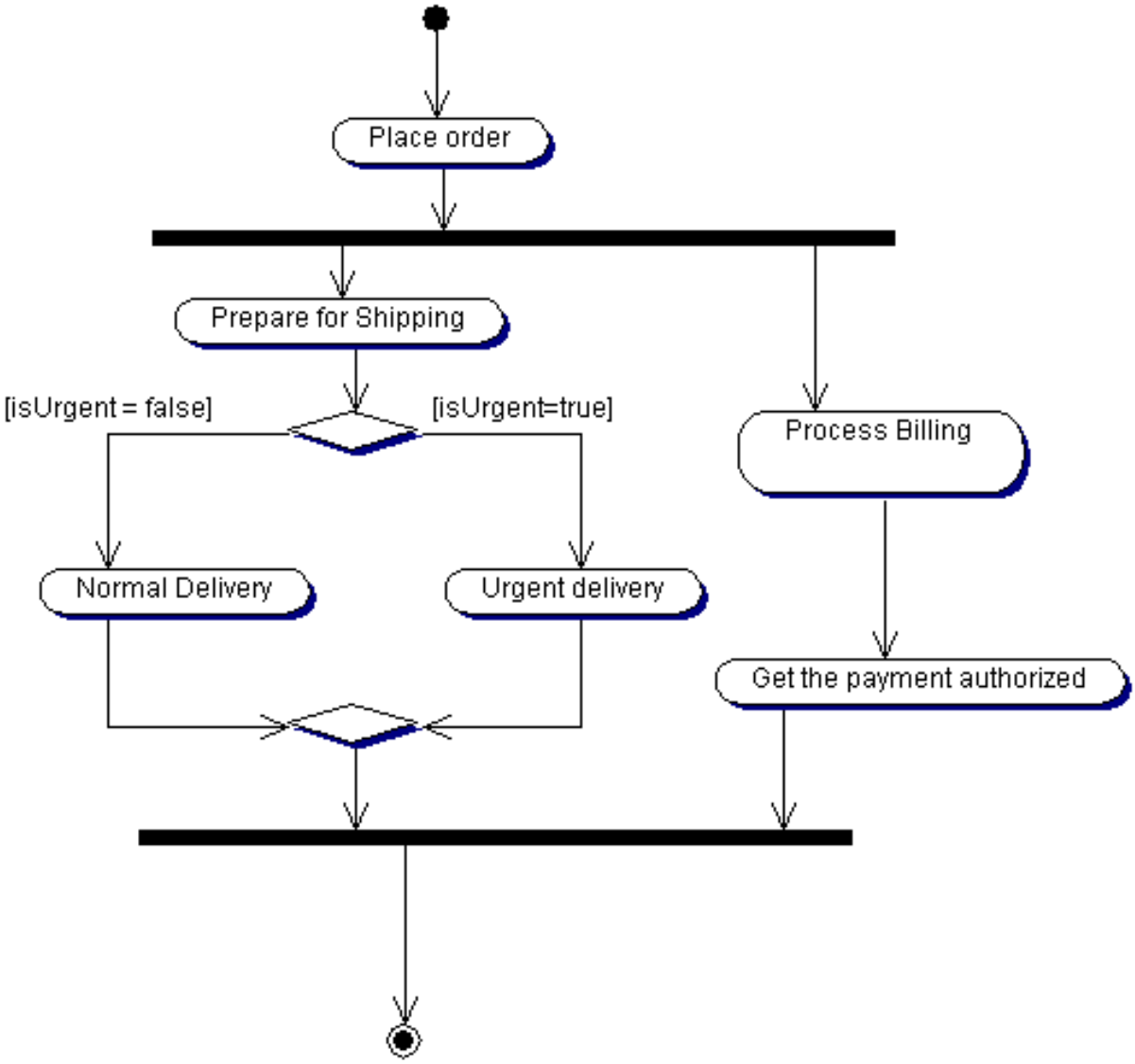


Activity diagram

An activity diagram is a special case of a statechart in which all or most of the states are activity states or action states and in which all or most of the transitions are triggered by completion of activity in the source states. You use it to show the flow from activity to activity within a system thereby neatly showing the workflow. A statechart diagram focuses attention on an object undergoing a process, whereas an activity diagram focuses on the flow of activities involved in a single process. It is an important diagram because it emphasizes the flow of control among objects. You can consider activity diagrams advanced versions of flow charts.

In the following example, the process begins with the dummy state marked by the black circle. The rounded rectangles represent the activities, and the arrows represent the flow between activities. After the activity Place order, where all the required details, such as shipping address and credit card details for completing the order are taken, the transition forks into two parallel activities, Prepare for shipping and Process billing. Finally, both parallel activities are joined into a single transition and end at the final dummy state represented by a concentric white/black circle. The fork and the join are represented by a solid bar.

When a transition branches out, the Guard expressions (such as [isUrgent=true] and [isUrgent=false]) label the transitions. Hollow diamonds indicate a branch into multiple transitions and the merge into a single transition.

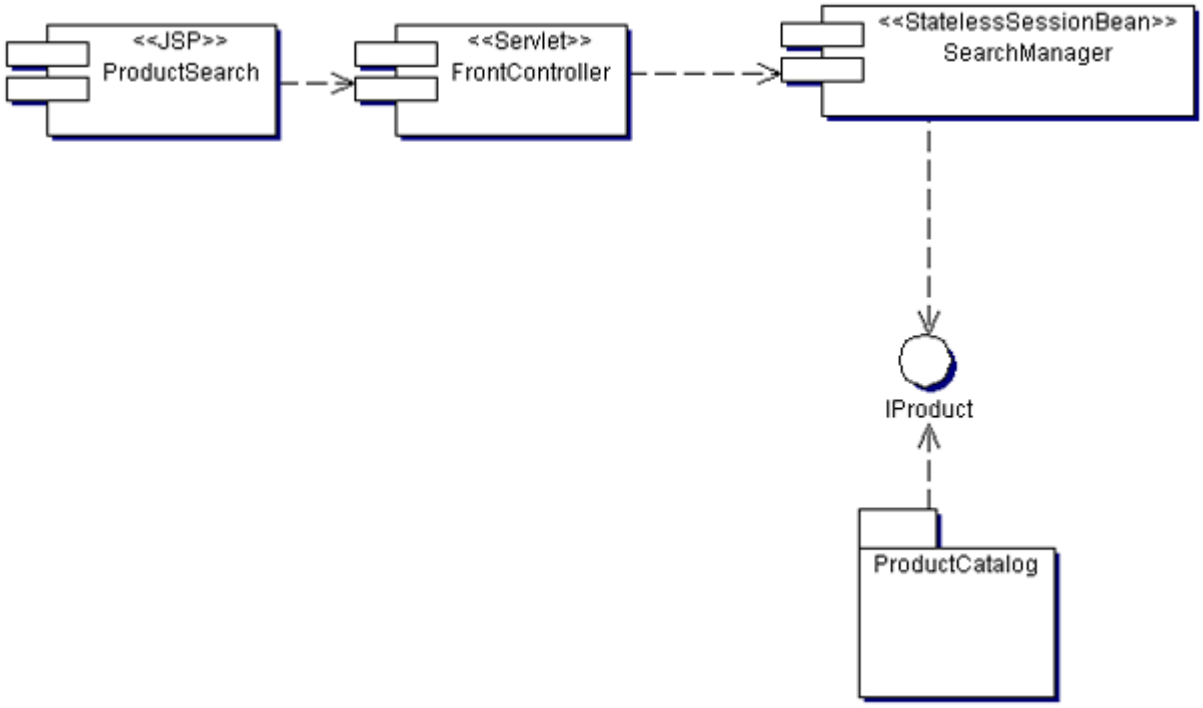


UML: Diagrams, continued

Component diagram

A component diagram shows the organizations and dependencies among a set of components. They address the static implementation view of a system by showing the dependencies among software components including source code components, binary code components, and executable components. A component diagram has only a descriptor form, not an instance form.

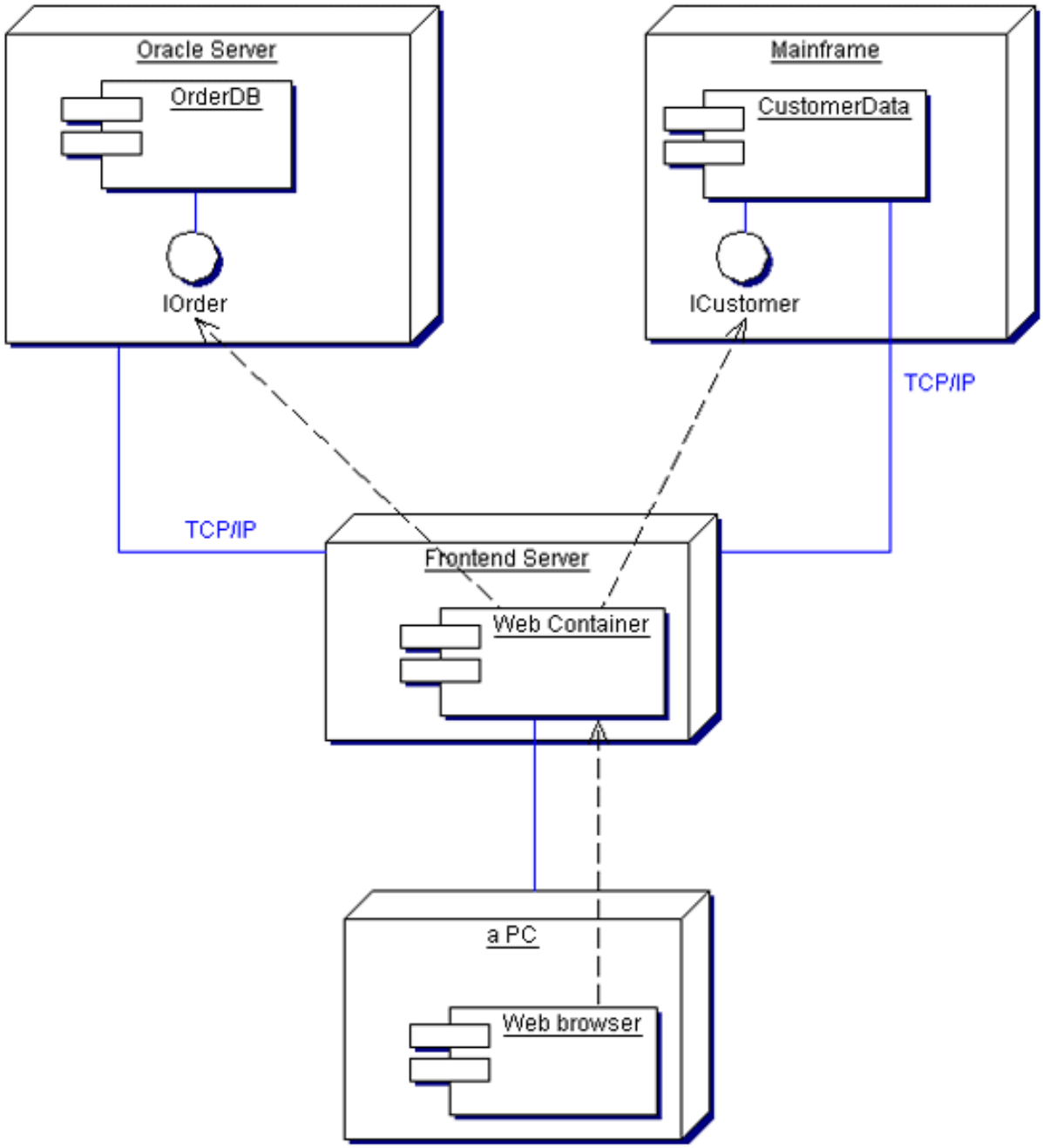
In the following example, a component is illustrated as a rectangle with two rectangles protruding from the left-hand side. The dashed arrow lines show the dependency between the components. A lollipop symbol represents the IProduct interface.



Deployment diagram

A deployment diagram shows the configuration of runtime processing nodes and the components that live on these nodes, and addresses the static deployment view of an architecture. Components that do not exist as runtime entities do not appear on this diagram; component diagrams represent them. The main difference between a deployment diagram and a component diagram is that the former shows the instances whereas the latter shows the definition of component types.

A cube represents the nodes, and an association link shows the physical connections between the nodes. In the following example, the node DB2Server represents the database node in which the OrderDB database component exists. Data access happens via the IOrder interface indicated by a lollipop symbol. The Web container component has a dependency with the IOrder, depicted by the dashed arrow line.



Encapsulation, inheritance, and use of interface

Encapsulation

Encapsulation is the localization of knowledge within a module. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but if the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten.

Encapsulation helps in:

- Distinguishing between the specification and the implementation of an operation.
- Modularity, which is necessary to structure complex applications designed and implemented by a team of programmers. It is also necessary as a tool for protection and authorization.

Inheritance

Inheritance is a process by which one object can acquire the properties of another object. There are two kinds of inheritance: class inheritance and interface inheritance.

Class inheritance creates a tight coupling between base and derived classes and is therefore often considered to break encapsulation. Also, because inheritance is a compile-time behavior, the implementation cannot be changed during runtime.

Based on this, the Gang of Four has defined two design principles. They are:

- **Program to an interface, not an implementation.**
  Clients only need to know the interface/abstract class that defines the interface. Hence, implementation dependencies between subsystems are avoided as the clients remain unaware of the classes that implement the interface.
- **Favor object composition over class inheritance.**
  As class inheritance exposes the parent classes, it can be replaced by object composition where a new functionality is obtained by assembling or composing objects to get more complex functionality. Composition respects the object's interfaces and consequently does not break encapsulation.

Summary

UML is a modeling language for specifying, modeling, and documenting object-oriented and component-based system architectures. Structural elements, behavioral elements, grouping elements, and annotational elements are the four basic UML building blocks for creating the models. You use these elements to draw the various UML diagrams that address the static and dynamic views of the system. Each diagram offers a unique perspective not available in other diagrams. While modeling a system, we often use a combination of different diagrams to address various concerns. It is important to remember that UML does not specify any process methodology; rather, it provides a modeling environment. You must utilize proven design principles like encapsulation and interfaces to create a robust and flexible system. Whenever possible, use interfaces to isolate clients from implementation, and use object composition instead of class inheritance.

For the exam, you must understand the different types of diagrams, so that given a

diagram, you can identify and interpret it.

Test yourself

Question 1:

Which two types of diagrams from the following can be derived from each other?

Choices:

- **A.** Sequence diagram
- **B.** Activity diagram
- **C.** Collaboration diagram
- **D.** Statechart diagram

Correct choice:

A and **C**

Explanation:

Choices A and C are the correct answers.

Sequence diagrams and collaboration diagrams are isomorphic (that is, one type of diagram can be converted into the other). Therefore, choices A and C are correct.

Activity and statechart diagrams do not have such relationships, so they are incorrect.

Question 2:

What is the relationship depicted by the following two classes?

Choices:

- **A.** Association
- **B.** Generalization
- **C.** Composition
- **D.** Strong dependency

- **E.** Aggregation

Correct choice:

C

Explanation:

C is the correct answer.

Composition, a stronger form of aggregation, is represented by a solid-filled diamond. Hence, choice C is correct.

Choice A is incorrect because association is simply represented by a line.

Choice B is incorrect because generalization is an inheritance relationship represented by a line with an open arrow at the end.

Choice D is incorrect because there is no such UML terminology as strong dependency.

Choice E is incorrect because aggregation, a special form of association that indicates a whole-part relationship, is represented by a line with an open diamond on the side of the whole part.

Section 3. Common architectures

Introduction

Architecture is a set of structuring principles that enables a system to consist of a set of simpler systems, each with its own local context independent of, but not inconsistent with, the context of the larger system as a whole. The architecture should not only satisfy the functional requirements but also the nonfunctional requirements. This does not happen often, however, because most of the efforts are targeted toward implementing the functionality, and other important quality attributes such as scalability, extensibility, and flexibility are conveniently forgotten. The architect's role should be to ensure that the architecture takes into account both functional and nonfunctional requirements. In this section, we discuss various quality attributes and how you can apply them to a tiered architecture.

Architectural characteristics (quality attributes)

As we've already mentioned, a system architecture should address both functional or business requirements and nonfunctional or service-level requirements. During

the initial phases, an architect has to define the quality of service measurement for each of the service-level requirements. Normally, there is a trade-off between these requirements. For example, to achieve better extensibility, an architect might introduce a modular architecture with many objects, thereby increasing the memory requirements, which impacts the performance. The service-level requirements of prime importance are:

- Scalability
- Maintainability
- Reliability
- Availability
- Extensibility
- Performance
- Manageability
- Security

Scalability

Scalability is the ability to accommodate more users with the required quality of service as the transactional load increases. A scalable system responds within acceptable limits even when there is an increased load. To scale a system that has met capacity, you can add hardware vertically or horizontally, thus leading to the following two types of scaling:

- **Vertical scaling**
  Vertical scaling is achieved by adding capacity (memory, CPUs, etc.) to existing servers. This requires only a few or no changes to the system's architecture. This type of scaling helps in increasing the capacity to serve more clients and requires less management of the resources. This type of scaling is comparatively easier and cheaper than horizontal scaling.
- **Horizontal scaling**
  Horizontal scaling is achieved by adding servers to the system. This increases the server's reliability, availability, and flexibility. Depending on the load-balancing algorithm, it sometimes also increases the performance. But horizontal scaling increases the complexity of the system architecture resulting in a decrease in manageability.
  J2EE architecture supports horizontal and vertical scaling. In horizontal scaling, the server can manage more components. In vertical scaling, with the help of load balancing and clustering, more clients can be served.

Maintainability

Maintainability is the ability to correct flaws in the existing functionality without impacting any other components. To enhance the maintainability, the system design

should be modular, and proper documents should be present for the system. This requirement cannot be measured during the deployment of the system.

Reliability

Reliability is the ability to ensure the integrity and consistency of the application and all of its transactions. You can increase reliability through the use of horizontal scalability, such as by adding more servers. The increase in reliability also increases the availability.

Architectural characteristics (quality attributes), continued

Availability

Availability is about assuring that services are available to the required number of users for the required proportion of time. The term 24x7 refers to the total availability. You can achieve total availability through a fault-tolerance mechanism such as replication. There are two types of replication, active and passive.

Extensibility

Extensibility is the ability to modify or add functionality without impacting the existing functionality. The key to an extensible design is to create an effective object-oriented (OO) design with low coupling, interfaces, and encapsulation. J2EE supports extensibility because it is component-based and allows you to separate an application's roles.

Performance

Performance is usually measured in throughput. Throughput is a system's response time. You can also determine performance using the number of transactions per unit time. By using load balancing and load distribution, you can increase the system's performance.

DNS Round Robin is a load-distribution algorithm, in which the first request is served by the first server and the subsequent requests are served by the subsequent servers. If there are five servers, then the sixth request is served by the first server. In load balancing, the request is served by the server with the lesser load.

You can also increase performance with efficient programming. For example, minimizing the number of network calls in the distributed application results in performance gains. Resource pooling and caching are also other ways of increasing performance.

Manageability

Manageability refers to the ability to monitor the system resources to ensure continued health of the system. A manageable system lets you dynamically configure the system without actually changing it.

Distributed applications are more complex to manage compared to monolithic applications, but distributed applications are more scalable, reliable, and available. So, it's a trade-off between manageability and other service-level requirements.

Security

Security ensures that information is not accessed, modified, or disclosed except in accordance with the security policy. A highly secure system is more costly and also harder to define and develop. The easier way to secure an application is by creating an architecture with separate functional components and applying security zones so that attacks are localized and impact is minimized, if they occur.

Security attacks generally try to compromise confidentiality and integrity of the system. Sometimes, they also take the form of denial of service (DoS) attacks that bring down a system by flooding it with messages. You can address security by using technologies such as firewalls, demilitarized zone (DMZ), data encryption, and digital certificates and methodologies such as good security policies and procedures.

Architectural tiers

The work done by any application program can be divided into four general functions:

- Data storage
- Data access logic
- Application logic
- Presentation logic

These functions can reside in a single tier or can be distributed across many tiers. Following are the different tiered architectures.

One-tier architecture or monolithic architecture

These applications are standalone applications that store data, apply business logic, and display results.

Two-tier architecture

This is a client/server architecture in which the user interface runs on the client, and the database is stored on the server. The actual application logic can run on either the client or the server. The drawback of this architecture is heavy load on the network due to heavy interaction between the client and server and also lack in modularity, which results in less flexibility.

N-tier architecture

Well-designed, distributed applications utilize n-tier architectures whereby each tier

is an autonomous unit you can develop and maintain separately, as technologies and business requirements change.

J2EE-based architectures consist of:

- **Client tier**
  The end user interacts with this tier. Clients can be thin clients, as in the case of browser-based applications, or fat clients, as in the case of client Java applications.
- **Web tier**
  This tier decouples the client tier from the business tier. Java servlets and Java ServerPages (JSPs) reside in this tier. Servlets act as controllers; they translate incoming requests and dispatch them to components that can invoke the necessary business events in the business tier. JSP pages combine static templates with dynamic data to create dynamic output the client tier uses for presentation to the user.
- **Business/application logic tier**
  This tier is generally implemented using Enterprise JavaBeans (EJBs) that act as business process objects and business domain objects. EJB containers provide various services, such as object distribution, persistence, transaction, resource management, security, and so on.
- **Enterprise information system (EIS) integration tier**
  The EIS integration tier interfaces between the business (and sometimes Web tier) objects and enterprise information systems. For example, data access objects (DAO) decouple enterprise beans (typically session beans or BMP entity beans) with enterprise data.
- **Enterprise information system (EIS) tier**
  This tier represents all the enterprise data and can be in many forms including relational databases, XML databases, and ERP systems. Well-architected and designed n-tier systems help in achieving all the nonfunctional service-level requirements of the system. The following table compares the various tiers in terms of the service-level requirements.

Service-level requirement	One tier	Two tier	N tier
Scalability	No	Not exactly	Highly scalable
Maintainability	Difficult to maintain. Rolling out changes to clients is also difficult.	Difficult as presentation and business concerns are intermixed.	Easier to maintain as application is layered. Also the clients do not need to be updated

				about new changes.
	Reliability	Single point of failure.	Database is still a single point of failure.	Can be designed to have fault tolerance using fail-over and redundant mechanism.
	Availability	Single point of failure.	Database is still a single point of failure.	Fault-tolerant mechanisms ensure 24x7 availability.
	Extensibility	Difficult.	Difficult as presentation and business concerns are intermixed.	Loosely coupled and hence better to extend.
	Performance	Good but might not perform well under heavy load.	Good performance only for small applications.	Good performance.
	Manageability	Easy to manage.	Not so easy to manage.	Difficult to manage.
	Security	Least robust as there is a single point of compromise.	Better than one tier.	Best security as security zones can be built around each tier; difficult to implement, however.

Summary

In this section, we discussed the basic definition of architecture and the service-level requirements. You learned that data storage, data access logic, application logic, and presentation logic are the four major functions for a system. Depending on the location where these concerns are addressed, you can classify the architecture as a one-tier, two-tier, or n-tier architecture. You also learned how the service-level

requirements relate to the number of tiers. For the exam, remember the definitions of various quality attributes and how they influence each other. Given a scenario, you should be able to point out the system's strengths and weaknesses.

Test yourself

Question 1:

Which of the following service-level requirements suffer in a one-tier system?

Choices:

- **A.** Manageability
- **B.** Extensibility
- **C.** Scalability
- **D.** Security
- **E.** Maintainability

Correct Choice:

B, **C**, and **E**

Explanation:

Choices B, C, and E are the correct answers.

One-tier systems are difficult to maintain because of the tight coupling between different concerns, such as presentation, business logic, and persistence. Changes to any one of them will affect the others. Because of this tight coupling, they are also not extensible. No tier separations exist, so they can only be vertically scaled if you add more resources to the system. However, this does not ensure a great performance under heavy load because horizontal scaling is not possible. Hence choices B, C, and E are correct.

One-tier systems are easier to manage because you don't have to manage as many components. Security might not be robust because there is a single point of failure, but it is still easier to implement in a one-tier system. Hence choices A and D are incorrect.

Section 4. Legacy connectivity

Introduction

A legacy system is a system that is used today but is based on an outdated system architecture. Monolithic mainframe-based systems, which are single-tiered, and client/server two-tier systems generally fall under this category. These applications cannot be scrapped completely due to the time and cost involved, and they must therefore coexist in a heterogeneous environment. Legacy connectivity focuses on integrating new e-business applications with the existing legacy systems, thereby extending the reach of legacy systems beyond its original goal. This section deals with how Java components can interact with such legacy systems.

Data-level integration

Data integration is the process of sharing or merging data from two or more distinct software applications to create a more highly functional enterprise application. Traditional business applications are highly data oriented -- they rely on persistent data structures to model business entities and processes. When this is the case, the logical approach is to integrate the applications by sharing or merging data.

Sharing or merging data is probably the easiest method for integration with a legacy application. Because the integrity checks present in user applications are bypassed, data corruption is possible. Hence, it is not suitable for applications where data integrity is critical. This method is also not suitable when complex data structures are involved or not much persistent data exists.

Java Database Connectivity (JDBC)

JDBC technology is an API that provides cross-DBMS connectivity to a wide range of SQL databases and access to other tabular data sources, such as spreadsheets or flat files. With a JDBC technology-enabled driver, you can connect all corporate data even in a heterogeneous environment.

Application-/business-/presentation-level integration

Application interface integration enables a higher level form of integration, where an application uses some of the functionality residing in other applications. This is achieved by using the APIs the applications expose. Typically, middleware such as message-oriented middleware (MOM), remote procedure calls (RPC), or object request brokers (ORB), is involved.

Java Message Service

The Java Message Service (JMS) is a messaging standard that allows application components to create, send, receive, and read messages. It enables distributed communication among J2EE components and legacy systems that can provide loosely coupled, reliable, and asynchronous messaging services.

Java IDL

Java Interface Definition Language (IDL) adds Common Object Request Broker Architecture (CORBA) capability to the Java platform, providing standards-based interoperability and connectivity. Java IDL enables distributed, Web-enabled Java applications to transparently invoke operations on remote network services using the industry standard IDL and Internet Inter-ORB Protocol (IIOP) defined by the Object Management Group. Runtime components include Java ORB for distributed computing using IIOP communication.

Simple Object Access Protocol

Simple Object Access Protocol (SOAP) is a wire protocol similar to CORBA's IIOP for communicating between applications running on different operating systems with different technologies and programming languages. It is an XML-based protocol that helps in calling an application, or even an individual object or method within an application, across the Internet via HTTP. Because HTTP is widely used and usually allowed by any firewall, there is a better chance for SOAP calls to be invoked through the firewall, which is not the case for IIOP and Remote Method Invocation (RMI).

Java Connector Architecture

The J2EE Connector Architecture (JCA) provides a Java solution to the problem of connectivity between the many existing application servers and EISs. By using the JCA, EIS vendors no longer need to customize their products for each application server. Application server vendors that conform to the JCA do not need to add custom code whenever they want to add connectivity to a new EIS.

Java Native Interface

The Java Native Interface (JNI) allows Java code that runs within a Java virtual machine (JVM) to operate with applications and libraries written in other languages, such as C, C++, and assembly. For example, a Java wrapper can be created for a legacy C++ application using JNI.

This method is complex and suitable only when you have access to the legacy system's source code.

Object mapping tools

You use object mapping tools to directly access the legacy system business logic and database tiers. Instead of using the existing legacy interface, underlying tiers are directly accessed. You use these tools to create proxy objects that access legacy system functions and make them available in an object-oriented manner. These tools are usually more effective than screen scrapers because they are not dependent on the format generated by the existing legacy interface.

Off-board server

An off-board server is a server that executes as a proxy for a legacy system. It

communicates with the legacy system using the custom protocols supported by the legacy system. It communicates with external applications using industry-standard protocols.

Screen scraper (terminal emulator)

A screen scraper component emulates a mainframe terminal. The screen scraper logs on to the mainframe like a normal user, sends requests to the mainframe, and then intercepts the character-based response of the mainframe. Therefore, it is useful for both CUI and GUI applications. The problem with a screen scraper is that even if there is a slight change in the application's behavior or the user interface, there is always the possibility that the screen scraper will stop working.

The screen scraper method is suitable when the legacy system does not expose any other programming interface and the source code is also not available.

B2B integration

In simple words, B2B commerce can be defined as business to business, or business conducted over the Internet. It is most commonly associated with buying and selling information, products, and services via the Internet or through the use of private networks shared among business partners. B2B can also be defined as the exchange of structured messages with other business partners over private networks or the Internet to create and transform business relationships. The exam asks a few questions about B2B architecture models, and they are:

- Spoke
- Exchange
- Hub

Spoke

A business is connected to the existing business partner's extranet as a spoke. A spoke is cheap and quick to implement. You just need a Web browser to read the data. The disadvantage is that the data is seen one partner at a time. So, it is difficult to assemble the big picture from other spokes.

Exchange

A B2B exchange solves many of the problems associated with relying on trading partner extranets. There is a central third-party marketplace that handles the infrastructure, and the partners supply and receive the required data. Then partners access the data through Web browsers. Although an exchange might represent a large number of trading partners, it is possible it might not represent all the company's trading partners. Also, the data may not be customized and can be limited.

Hub

A B2B hub serves as a single point of control effectively integrating business processes and information across a business ecosystem. On its own hub, a company aggregates the demand from its entire network, including strategic partners, exchanges, and internal relationships. The advantage is access to rich customized data from the partners, but the disadvantage is that it requires a lot of investment.

Summary

In this section, we discussed the definition of a legacy system and the different ways to interact with the system. We concluded with B2B integration scenarios. In the exam, you might be presented with a scenario and asked to choose the best integration method. So, you must understand each possibility and determine which one is the best given the constraints imposed by the problem.

Test yourself

Question 1:

What is an off-board server?

Choices

- **A.** A screen scraping program
- **B.** A server that runs in a demilitarized zone
- **C.** A proxy for a legacy system
- **D.** A Web proxy server

Correct choice:

C

Explanation:

Choice C is the correct answer.

An off-board server is a server that executes as a proxy for a legacy system. Hence, choice C is correct.

Section 5. Enterprise JavaBeans

Introduction

EJB is a component architecture prescribed by the J2EE technology for the development of component-based distributed applications. Features such as transaction management, state management, resource pooling, and security are automatically available to the business components written using EJB. In an n-tier architecture model, EJB components reside in the middle tier (typically in an EJB container hosted by the application server) and contain the business logic for the applications. EJB components help in rapid and simplified development of distributed, transactional, secure, and portable applications based on Java technology. However, EJB is not a silver bullet for all your problems. You should only use EJB technology if an application requires security and/or transaction support.

As mentioned earlier, the exam covers only EJB 1.1 and not the latest versions.

EJB component model contract

The EJB component model requires:

- Home interface
- Remote interface
- Bean class
- Primary key class

Home interface

The home interface extends `javax.ejb.EJBHome`. The home interface allows the client to create and remove the beans. For entity beans, this interface also allows you to find an existing bean.

Remote interface

The remote interface extends `javax.ejb.EJBObject` and defines the bean's business methods the client can call.

Bean class

The bean class extends `javax.ejb.EntityBean` for entity beans and `javax.ejb.SessionBean` for session beans. This class implements the bean's business methods. It is important to note that the bean class does not implement either the home interface or the remote interface of the bean.

Primary key class

The primary key class is present only for an entity bean; however, it is optional. This

class implements `java.io.Serializable` and contains one or more public fields whose names and types match a subset of container-managed fields in the bean class. This class can remain undefined until deployment.

EJB types

There are two types of enterprise beans.

- Session beans:
  - Stateful
  - Stateless
- Entity beans:
  - Container-managed persistence
  - Bean-managed persistence

(**Note:** In EJB 2.0, there is one more type of EJB called message-driven beans for asynchronous messaging.)

Session beans

Session beans represent business logic, rules, and workflow. They can exist as stateful or stateless session beans. At deployment time, the type of session bean is revealed to the container with the help of an identifier in the deployment descriptor.

- **Stateful session beans**
  Stateful session beans can maintain the conversational state of a client across methods and transactions and are therefore dedicated to the same client for their lifetime. A conversational state simply involves maintaining the reference to the client's prior states. An online shopping cart is the best example of maintaining a conversational state where the contents of the client cart should be kept until the client's session is active. The use of stateful session beans is the correct choice for modeling this component.
- **Stateless session beans**
  On the other hand, stateless session beans cannot maintain conversational state. This does not mean that stateless session beans cannot hold any state information, but that such stored information cannot be client specific. These are lightweight objects, and a minimal number of instances can be swapped and reused for many clients. Thus, the performance is better compared to stateful session beans.

Stateless or stateful?

Choosing between stateful and stateless beans depends on whether you need to maintain the conversational state of the client. Stateful session beans are instantiated on a per-client basis and can multiply and consume resources rapidly. So, if you don't need to maintain state, then it's always better to use stateless beans for performance reasons. Also, remember that if the conversational state must persist even after the client is gone, then entity beans are a better choice for storing this data. And because stateless session beans are never passivated like stateful session beans, they offer better performance than stateful session beans.

Entity beans

Entity beans represent the data stored in a data store, such as a database, file system, nontrivial storages like LDAP, or any form of persistent storage. For the sake of simplicity in this tutorial, whenever we refer to a database, it means any kind of persistent data store.

Multiple clients can access the same bean, and the container manages the concurrency. These beans are persistent across client sessions.

An entity bean can be classified into two types depending on whether the container or the bean developer handles the persistence logic.

- **Container-managed persistence (CMP)**

  The EJB container handles all the database access required by the entity bean. The bean class does not contain any database-related code, and it is not tied to any particular database. The bean can be deployed to any J2EE server, with any database.

  CMP benefits include:

  - Portability across database schema and vendors.
  - Absolutely no code (like SQL) for data access; hence, the bean looks cleaner.
  - Faster development time because the persistent logic is handled behind the scene.
  - The persistent code generated by the container is generally optimized, and the bean is consequently better quality.

  The costs of CMP use involve:

  - The generated queries cannot use the native features provided by the database vendor. For example, using a particular index or hint with an Oracle database is not possible.

- **Bean-managed persistence (BMP)**

Vendors might not provide mapping tools for some nontrivial data stores like LDAP or XML databases. BMP is the only solution in this case. Here, the bean developer writes the logic for loading and storing the data to a backing store. In the case of relational databases, this is typically done using JDBC, potentially through the use of data access objects.

Transaction management

A transaction is a unit of work that either fully completes (a commit), or is not completed at all (a rollback). A transaction adheres to the following "ACID" principles.

Atomicity

This property ensures that the transaction is performed as a single unit of work where everything is totally complete or everything remains untouched. Imagine if you make a purchase through an online store and you are charged for the items by the payment bean, but the order bean doesn't ship you the goods because of a failure. The atomicity property avoids such a scenario by ensuring that both the payment and shipping happens or the payment reverses if shipping fails.

Consistency

A consistent transaction should leave the data store in a consistent state. In the previous example, if you are charged for the items by the payment bean, and your order ships but the order bean doesn't update the record, then the database is in an inconsistent state. The consistency property ensures that this does not happen.

Isolation

This property ensures you do not see the other running transactions in the database, thereby preventing one transaction from corrupting the other.

Durability

Durable transactions should survive a system failure. If the system fails in the middle of the transaction, then the system should back out the transaction for consistency and atomicity.

Transaction management, continued

EJB transactions can be implemented in two ways:

- Container-managed transactions (CMT)
- Bean-managed transactions (BMT)

Container-managed transactions

CMT is a declarative transaction demarcation, where the transaction-related details are written into the bean's deployment descriptors. The container takes care of the actual transactions. Nested or multiple transactions are not allowed in the enterprise beans. Typically, the container begins a transaction immediately before an enterprise bean method starts. It commits the transaction just before the method exits. Each method can be associated with a single transaction.

Container-managed transactions do not require all the methods to be associated with transactions. When deploying a bean, you specify which of the bean's methods are associated with transactions by setting the transaction attributes.

To specify transaction requirements for a method, you must specify a transaction attribute. These attributes control the scope of a transaction. There are six possible values for the transaction attributes:

- Required
- RequiresNew
- NotSupported
- Supports
- Mandatory
- Never

Required

This bean method must be part of a transaction. If called outside a transaction, a new transaction is automatically started. Otherwise, the method uses the existing transaction.

RequiresNew

When this bean method is called, a new transaction is always started, and the existing transaction is suspended.

NotSupported

The transactional context of the calling client is not propagated to the enterprise bean. Instead, the client transaction is suspended, and the bean method runs within an unspecified transaction context.

Supports

This method can be called independently or as part of a transaction. If the client is associated with a transaction context, the bean runs within the same transaction context. Otherwise, the bean method runs within an unspecified transaction context.

Mandatory

This bean method can be called only as part of a transaction. If the client invokes the method without a meaningful transaction context, the container throws `TransactionRequiredException`. Otherwise, the method runs within the client's transaction context.

Never

This bean method cannot be called as part of a transaction. If the client invokes the method with a meaningful transaction context, the container throws a `java.rmi.RemoteException`. Otherwise, the bean method runs within an unspecified transaction context.

Bean-managed transactions (BMT)

BMT is a programmatic transaction demarcation, and the programmer should write the code in the enterprise bean and handle the transactions. BMT can use either JDBC transaction or JTA transaction. The transaction manager of the DBMS provides JDBC transaction. There is a possibility that this transaction manager might not work with the heterogeneous databases. The J2EE transaction manager provides JTA transactions, and the transaction can span and update multiple databases of different vendors.

A session bean can use either BMT or CMT, whereas an entity bean can use only CMT.

Using data access objects

You use DAO to decouple the business logic and the data access logic. In entity beans, if you use BMP, then the code should contain the business logic as well as data access logic. Instead of having both the concerns in the same place, you can move the data access logic to a DAO class. In that case, data access logic is hidden from the entity bean. It is a modular and reusable way of coding, and you can easily swap and move the DAO to another database. Later, if you must convert the BMP bean to a CMP bean, this task becomes easier because of the segregation of the concern.

The DAO also has some cons. It creates one more layer, and the programmer is responsible for properly creating and garbage collecting DAO. Also, the programmer must write good SQL queries.

Security

It is important to know that EJB focuses on authorization rather than on authentication. So, you normally specify who can access which methods and not how the users authenticate themselves to the system. Authentication mechanisms are generally defined by the container vendor.

The EJB architecture encourages the programmer to implement the enterprise bean

class without hard-coding the security policies and mechanisms into the business methods. In most cases, the enterprise bean's business method does not contain any security-related logic. The deployer configures the security policies for the application depending on the needs of the target environment.

EJB offers two kinds of security:

- Declarative
- Programmatic

Declarative

The security rules are defined declaratively in the bean's deployment descriptor. Roles are defined and then the method permissions are defined in the descriptor. Method permissions indicate which roles are allowed to invoke which methods. During the deployment, the deployer maps these abstract roles to actual users in the target system.

Programmatic

In some cases, the declarative security is not enough. For example, a bean might want to restrict access based on the incoming user's role and the business method parameters. You use the `getCallerPrincipal()` and `isCallerInRole()` methods to programmatically authorize the access. The `getCallerPrincipal()` method returns the enterprise bean's caller, and you use the `isCallerInRole()` method to get the caller's role.

Summary

In this section, we examined the EJB programming model and the required classes/interfaces to build a bean. We then discussed the different types of beans along with their uses. We also reviewed whether to choose stateless or stateful bean for a given task and looked into the persistence and transaction possibilities provided by the EJB model. Lastly, we considered the use of DAO to write better data access code and explored the EJB security model.

Test yourself

Question 1:

A nontransactional client accesses an EJB and a `TransactionRequiredException` is thrown. What is the transaction attribute of this EJB?

Choices:

- **A.** Required

- **B.** Mandatory
- **C.** RequiredAlways
- **D.** RequiredNew

Correct choice:

B

Explanation:

Choice B is the correct answer.

An EJB method marked with a Mandatory transaction attribute can be called only as part of a transaction. If the client invokes the method without a transaction context, the `TransactionRequiredException` is thrown. Hence, choice B is correct.

Choices A and D are incorrect because these attributes do not throw an exception under the given scenario.

Choice C is incorrect as there is no such transaction attribute as RequiredAlways.

Question 2:

Which of the following statements are true?

Choices:

- **A.** Stateful session beans can maintain client-specific state.
- **B.** Stateless session beans can maintain state, but it will not be specific to a client.
- **C.** Stateful session beans extend `javax.ejb.StatefulSessionBean` whereas stateless session beans extend `javax.ejb.SessionBean`.
- **D.** A stateful session bean cannot use bean-managed transaction.

Correct choice:

A and **B**

Explanation:

Choices A and B are the correct answers.

A stateful session bean is dedicated to a client during its lifetime. Therefore, you can use a stateful session bean to store client-specific state, so choice A is correct.

A stateless session bean can have an instance variable that stores state information. But because the bean instance is not dedicated to a client and is swapped across multiple clients in its lifetime, the stored information cannot be specific to a particular

client. For example, you cannot store a user's shopping cart in the instance variable and expect to fetch it later. Rather you can use it to store generic data like information about the database, JNDI names, and so on. Thus, choice B is correct.

A session bean, regardless of its type, always extends `javax.ejb.SessionBean`. Hence, choice C is incorrect.

Session beans can use both container-managed and bean-managed transactions. So, choice D is incorrect.

Section 6. Enterprise JavaBeans container model

Introduction

The EJB container provides the runtime support for the deployed EJB instances. From the perspective of the enterprise beans, the container is a part of the target operational environment. The container runtime provides the deployed enterprise beans with transaction and security management, network distribution of clients, scalable management of resources, and other services generally required as part of a manageable server platform. The container provides a simple, standard API between the enterprise bean and the container. This API is called the EJB component contract. In this section, let's see why a container pools the bean instances and passivates the bean. We also discuss the container's lifecycle management capabilities and how system monitoring helps in a healthy system.

Bean instance pooling

EJB components are heavyweight objects with many classes (programmer defined and system generated) operating under the hood. Creating and removing them frequently is consequently an expensive operation. To avoid this, an EJB container uses a pool of instances shared between users. Pooling is done only for stateless beans and entity beans; stateful beans need to maintain state and thus cannot be swapped between users.

The benefits of pooling are:

- Timely handling of more requests because time is not wasted in creating/deleting objects.
- Complete transparency to the client.
- Declarative mechanism that does not require any change in the bean code and can be fine-tuned based on the resources in hand and the number of requests.

Bean passivation

To efficiently manage the resources, an EJB container transfers the state of an idle bean instance to some form of secondary storage. The transfer from the working set to secondary storage is called instance passivation. The transfer back is called activation. This is done only for stateful session beans and entity beans. For a stateless session bean, passivation is not needed, as there is no state to preserve. The container can simply create another stateless session bean instance, if one is needed, to handle an increase in client workload.

The benefits of passivation include:

- Handling more requests with fewer resources by passivating idle instances.
- Complete transparency to the client although a delay might be experienced while activating the instance.
- Declarative mechanism that does not require any change in the bean code and can be fine-tuned based on the resources in hand and the number of requests.

It is important to know that a stateful session bean cannot be passivated if it is in a transaction.

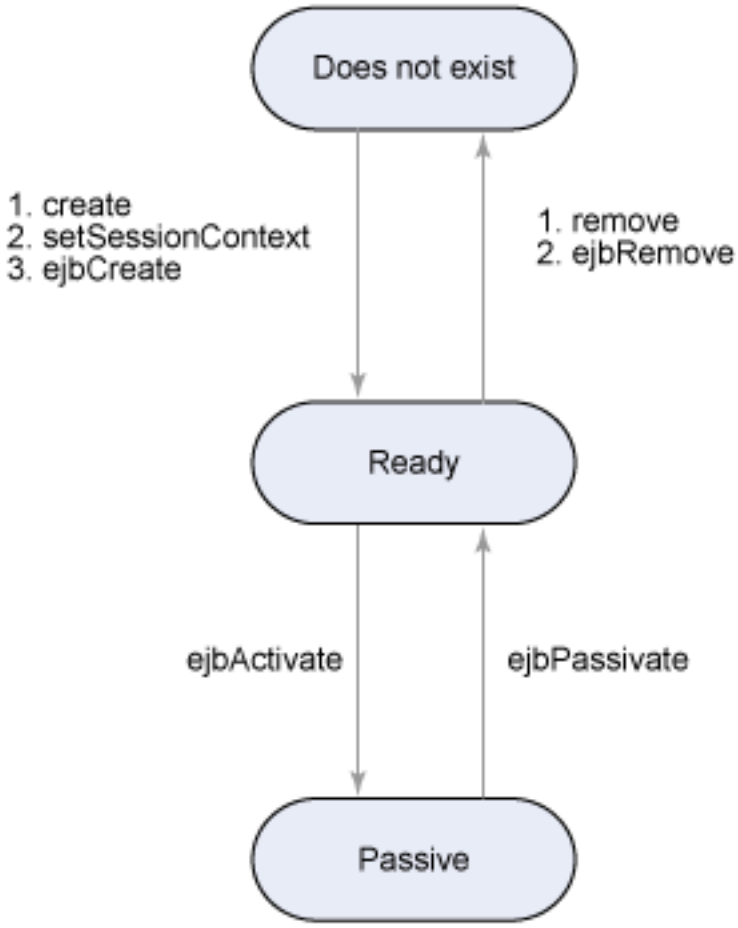
Lifecycle management

The lifecycle of an Enterprise JavaBean is managed by the container that ensures optimal utilization of resources by mechanisms, such as instance pooling and passivation. Because of this, lifecycle management is essential for the scalability of an application. If you have chosen a proper Enterprise JavaBean for your task and configured the correct parameters for pooling and passivation behaviors, you can rest assured that the application will scale well.

The lifecycle varies depending on the type of Enterprise JavaBean. Let's examine the beans types one by one.

Stateful session beans

The following diagram illustrates the lifecycle of a stateful session bean. The client initiates the lifecycle by invoking the `create()` method. The EJB container instantiates the bean and then invokes the `setSessionContext()` and `ejbCreate()` methods in the session bean. The bean is now ready to serve the clients.

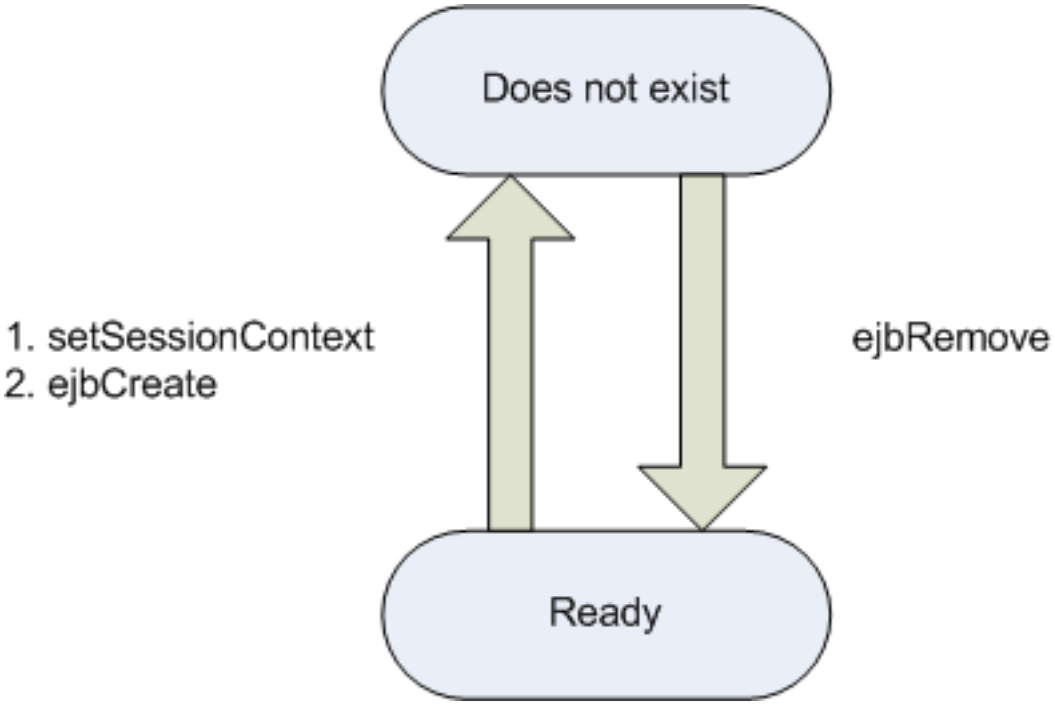


In the ready stage, the EJB container might decide to passivate the bean by moving it from memory to secondary storage to preserve some precious memory. Just before passivating, the EJB container calls the `ejbPassivate()` callback method to notify the bean. The bean can perform any cleanup action such as closing any open JDBC actions. If a client call comes when the bean is in passive stage, the EJB container activates the bean and moves to the ready stage. Just before the bean is available to the client, the `ejbActivate()` method is called so that the bean can initialize any resources closed during passivation.

At the end of the lifecycle, the client invokes the `remove()` method, and the EJB container calls the bean's `ejbRemove()` method. The bean's instance is now ready for garbage collection.

Stateless session beans

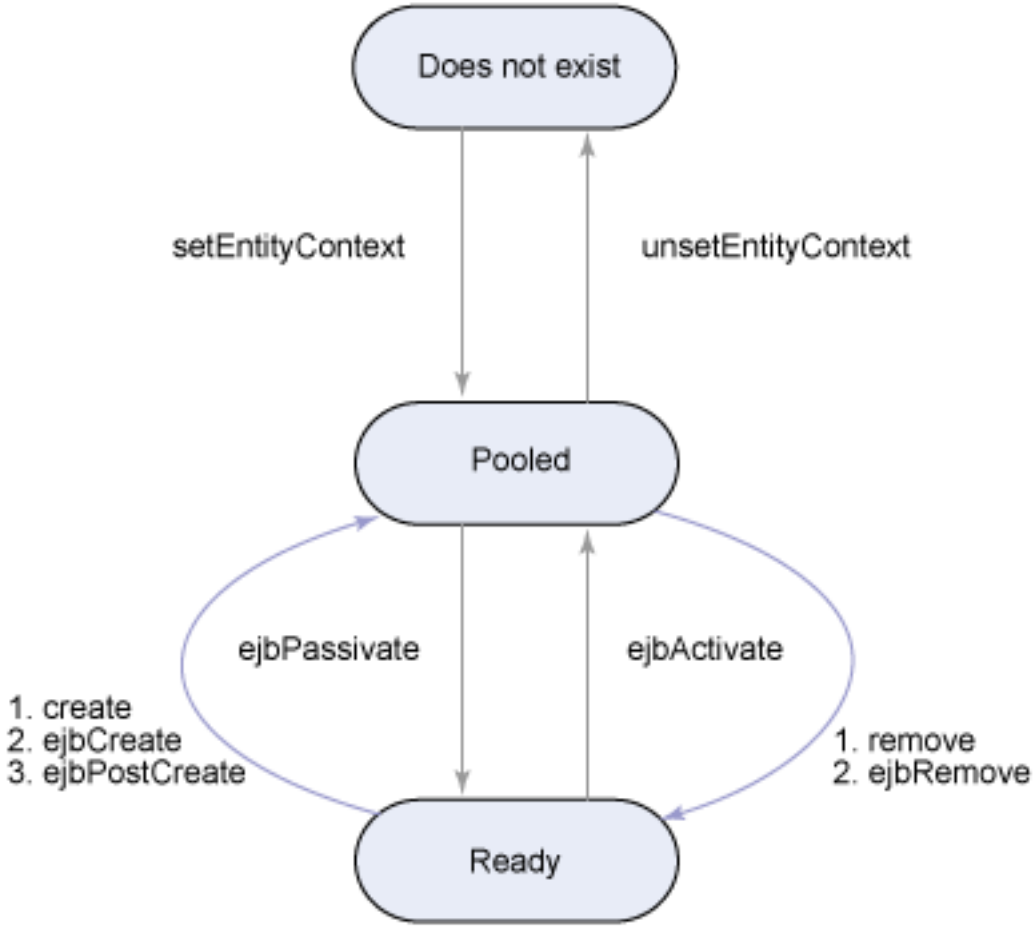
Because there is no state maintenance, stateless session beans contain a simple lifecycle with only two possible states during their lifetime. The following diagram illustrates the events that happen in the lifecycle of a stateless session bean.



Lifecycle management, continued

Entity beans

The following diagram illustrates the lifecycle of an entity bean. After the EJB container creates the instance, it calls the entity bean class's `setEntityContext()` method. The `setEntityContext()` method passes the entity context to the bean.



Once instantiated, the entity bean moves to a pool of available instances. In the pooled stage, the instance is not associated with any particular EJB object identity. All instances in the pool are identical. This is the reason why the container can pool the resources between clients. The EJB container assigns an identity to an instance when moving it to the ready stage.

There are two paths from the pooled stage to the ready stage. The client invokes the `create()` method, causing the EJB container to call the `ejbCreate()` and `ejbPostCreate()` methods. In the second path, the EJB container invokes the `ejbActivate()` method. While in the ready stage, an entity bean's business methods may be invoked.

Similarly, there are two possible paths from the ready stage to the pooled stage. First, a client may invoke the `remove()` method, which causes the `ejbRemove()` method to be invoked. Second, the EJB container may invoke the `ejbPassivate()` method to conserve resources.

At the end of the lifecycle, the EJB container removes the instance from the pool and invokes the `unsetEntityContext()` method. It is important to know that the `ejbRemove()` method is not invoked when the instance is removed from the pool, as in session beans, because this would delete the entity data permanently from the data store.

System monitoring

Every EJB container (and J2EE application server) allows an administrator to perform system monitoring of the quality of service requirements and enables them to declaratively and dynamically change the system configuration such as the minimum/maximum number of beans in the pool, concurrency parameters, and transaction levels without affecting the system. This increased manageability helps in improving the application performance by ensuring optimal configuration of server resources for the present load.

Summary

In this section, we discussed the role of the EJB container, instance pooling, and instance passivation, and their benefits. We also looked at the lifecycles of various types of beans and explained how they help in increasing the scalability of the application under load. Finally, you learned how system monitoring helps in ensuring the proper functioning of the applications and the server.

For the exam, remember that stateless session beans and entity beans can be pooled whereas stateful session beans and entity beans can be passivated. Also, understand the benefits of bean pooling and bean passivation. It is equally important to understand the lifecycle of entity beans, stateless session beans, and stateful session beans.

Test yourself

Question 1:

For which of the following beans can instance pooling be performed?

Choices:

- **A.** CMP entity bean
- **B.** BMP entity bean
- **C.** Stateless session bean
- **D.** Stateful session bean

Correct choice:

A, **B**, and **C**

Explanation:

Choices A, B, and C are the correct answers.

Stateful session beans cannot be pooled because the instances need to maintain state and thus cannot be swapped between users. Hence, choice D is incorrect.

Apart from this, stateless session beans can be pooled because there is no need for state maintenance. Entity beans can also be pooled because their state information is always backed up in the persistent store and can be retrieved back. Therefore, choices A, B, and C are correct.

Section 7. Protocols

Introduction

In this section, we discuss HTTP, HTTPS, IIOP, and JRMP per the exam objectives. HTTP and HTTPS are targeted for serving Web pages, whereas IIOP and JRMP act as the transport layer for CORBA and RMI, respectively. We also see how you can use HTTP/HTTPS to tunnel other protocols when they are blocked.

Hyper Text Transfer Protocol (HTTP)

HTTP is a protocol for moving hypertext files across the Internet. It requires an HTTP client program, such as a Web browser on one end and an HTTP server program on the other end. HTTP is the most important protocol used in the World Wide Web. If you are reading this tutorial on the Internet, you are probably already using the HTTP protocol.

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, Uniform Resource Identifier (URI), and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta-information, and possible entity-body content. HTTP communication usually takes place over TCP/IP connections. The default port is TCP 80, but you can use other ports as well.

Over the years, the protocol has evolved from versions 0.9 and 1.0 to the present 1.1. The primary difference between HTTP 1.0 and HTTP 1.1 is the way in which they handle the connections. HTTP 1.0 uses a tear-down approach where a new connection is used for each request/response exchange. For example, if a page refers to two inline images, one style sheet, and one JavaScript file, then the client needs to open five (one for the main page, two for inline images, one for the style sheet, and one for JavaScript) separate TCP connections to the same server, thereby increasing the load on the HTTP servers and causing congestion on the

Internet. In HTTP 1.1, a persistent (also referred to as keep-alive) connection is used for one or more request/response exchanges, although connections might be closed for a variety of reasons.

An HTTP request can be of GET, POST, HEAD, PUT, OPTIONS, DELETE, TRACE, or CONNECT types. Of these, only the following three are widely used.

GET

Retrieves whatever information (in the form of an entity) is identified by the request URI. The method is triggered when the user directly types the URL in the browser's address bar, when the user clicks a link, or when the FORM specifies the GET method using the method attribute.

POST

Uses the body of the request to send the data and is useful when posting secure information, such as login data. This method allows the client to send data of unlimited length to the Web server a single time. It can also be used to upload both text and binary data.

HEAD

Used to retrieve the meta-information (HTTP headers) about a resource. Proxy servers often use it to identify whether a fresh update is available for a locally cached resource by verifying appropriate HTTP headers.

The pros of HTTP use are:

- Simple to implement and extend
- Ubiquitous standard
- Most of the firewalls do not block HTTP traffic

The cons of HTTP use are:

- Stateless: Each request/response is an isolated, self-contained unit. The application's previous state is not known to the current request. This is all right for simple Web pages, but state maintenance is of prime importance for many dynamic Web applications. However, this issue can be resolved with:
  - Cookies
  - URL rewriting
  - HTTPS
- Insecure: Because all requests/responses are in clear text, it is easy for a determined person to sniff the traffic to know the communication details. Also, there is no verification about the client and the server. These days when it's easy to spoof a DNS record, you could accidentally hand over

your sensitive information to a hacked server that falsely claims to be a genuine server.

Hyper Text Transfer Protocol, Secure (HTTPS)

Secure Sockets Layer (SSL) is a protocol developed by Netscape for transmitting documents securely over the Internet. SSL works by using a private key to encrypt data transferred over the SSL connection. HTTPS is the HTTP protocol using SSL for its transport. The default port is 443.

At the start of the connection, an SSL handshake process takes place where the identities of the client (optional) and server are verified. Following this, based on the capabilities of both the parties, symmetric session keys are generated. Then the handshake is completed and the secure session begins. The client and the server then use these session keys to encrypt and decrypt the data they send to each other and to validate their integrity.

The pros of HTTPS use are:

- Secure: Only the sender and the receiver can see the data. Any intermediary gets to see only encrypted data that is meaningless unless the key to decrypt it is known. Also, both the parties are verified, and hence there is no chance of confidential data getting into wrong hands.
- State maintenance: State is maintained between subsequent requests. An application can use this to maintain client-specific details.
- Firewall support: Like HTTP, HTTPS is also allowed through most firewalls.

The cons of HTTPS use are:

- Slower and expensive: The mathematics used for encryption/decryption requires more processing power. There is also an increased amount of data exchanged between the two parties. Both of these factors slow down the performance. The protocol also requires a digital certificate for the server which means you must spend money to buy it from a certified authority and must renew it when it expires.

Internet Inter-ORB Protocol (IIOP)

IIOP, which is a critical part of CORBA, is an object-oriented protocol that allows distributed programs written in different programming languages to communicate over the Internet. IIOP enables two or more object request brokers (ORBs) to cooperate to deliver requests to the proper object. CORBA and IIOP enable applications to cross boundaries of different computing machines, operating systems, and programming languages.

Starting with CORBA 2.3, pass-by-value is supported, and applications can therefore transfer objects from the server to the client by copying. Prior to version 2.3, CORBA only supported remote references, and the lack of object transferring was seen as a major disadvantage.

Java applications can use IIOP in two ways, either by using Java IDL or RMI over IIOP. Although using Java IDL is the most direct solution, the RMI/IIOP solution developed by Sun Microsystems and IBM is better in terms of the simplicity in coding for accessing CORBA services. RMI/IIOP combines RMI's ease of use with CORBA's cross-language interoperability. It also provides a convenient way to use both Java Remote Method Protocol (JRMP) and IIOP on the same network, thus offering flexibility for customers with mixed environments.

It's worth noting that no default port for IIOP exists, as the ports are dynamically assigned when an object server binds to a name. Because of this, IIOP supports tunneling over the HTTP protocol to overcome firewall rules.

The pros of IIOP are:

- Interoperable standard: Allows programs at different locations and developed by different vendors to communicate.
- Wide range of standard services, such as naming, security, and transactions.

The cons of IIOP are:

- Performance: Because the calls are remote calls, there is a heavy penalty on performance.

Java Remote Method Protocol

JRMP is the native wire protocol used for RMI. It is a connection-based and stateful protocol. Similar to IIOP, it can invoke methods, pass arguments, return values, and pass objects and exceptions over the network. For passing objects, JRMP uses the Java object serialization mechanism.

JRMP supports naming service which runs at the default port of 1099. The object servers are dynamically assigned the port number, as in IIOP.

JRMP clients have built-in support for HTTP tunneling. They use the HTTP POST method for this purpose.

RMI/IIOP is another wire protocol for RMI. It is interoperable with CORBA services and therefore the most desired wire transfer protocol for RMI. A J2EE application server (as well as an EJB server) is required to provide support for the RMI/IIOP protocol.

The pros of JRMP are:

- Simple and high performance: The protocol is simple without the complexities in IIOP, so it is much easier to use, which results in better performance.

The cons of JRMP are:

- Firewall support: Although HTTP tunneling is provided for JRMP clients, the JRMP servers need to install additional proxy plug-ins, such as RMIServlet, to handle the tunneled calls.
- Services: The only service available is the naming service. On the other hand, a CORBA implementation provides many services like security, transaction, concurrency, and persistence.
- Interoperability: Works only in a Java environment. You can use JNI to provide interoperability with other languages, but it is not available out of the box.

Usage scenarios

The following table lists the possible usage scenarios.

Protocol	Can be used when
HTTP	• Unstructured data, such as Web pages or MIME types, such as video files, need to be transferred. • Session state need not be maintained. • Data security or party verification is not expected.
HTTPS	• Unstructured data, such as Web pages or MIME types, such as video files, need to be transferred. • Session state needs to be maintained. • Data confidentiality and party verification is required.
IIOP	• CORBA model is used with mixed environments requiring platform and language independence. • Everything is inside the same network (might need tunneling for access from/to outside).

JRMP	• RMI model is used. • Everything is developed in Java language. • Everything is inside the same network (might need tunneling for access from/to outside). • A free or low-cost implementation is required. • Code portability and garbage collection are required.
RMI/IIOP	• RMI model is used. • Simplicity of Java language and interoperability with CORBA is required. • Everything is inside the same network (might need tunneling for access from/to outside), • Both JRMP and IIOP must be used on the same network.

Firewall and HTTP tunneling

Because HTTP and HTTPS are widely used, firewall administrators generally do not block HTTP/HTTPS default ports. However, this is not the case for IIOP and JRMP protocols that use dynamic ports for their object servers. Thus, they get blocked and must use an HTTP tunneling mechanism to go through the firewall.

HTTP tunneling works by encapsulating the required protocol as HTTP requests and responses. HTTP tunneling should be the last resort because it does not yield good performance and requires extra setup and support.

Summary

In this section, you learned about the HTTP protocol and the differences between the various versions. We then discussed the HTTPS protocol along with its pros and cons. CORBA's IIOP protocol was also explored followed by the native RMI protocol, JRMP. We also delved into the advantages of using RMI/IIOP over JRMP. We listed the usage scenarios of various protocols, and finally, you saw the issues to remember when dealing with a firewall.

For the exam, you should remember the features of each protocol so you can select the correct protocol for a given scenario. Also, understand how a firewall could affect the protocol you plan to use and the possible solutions, such as HTTP tunneling, to

overcome that.

Test yourself

Question 1:

Which of the following protocols help in identifying the parties?

Choices:

- **A.** HTTP
- **B.** IIOP
- **C.** RMI/IIOP
- **D.** HTTPS
- **E.** JRMP

Correct choice:

D

Explanation:

Choice D is the correct answer.

SSL helps in encrypting the data and identifies the sender to the receiver and vice versa. Thus, the HTTPS protocol, which is HTTP over SSL, provides this feature. So, choice D is correct.

The remaining protocols do not identify the parties and are therefore incorrect.

Question 2:

Which of the following is the standard port for a JRMP object server?

Choices:

- **A.** 80
- **B.** 443
- **C.** 8080
- **D.** 1099
- **E.** None of the above

Correct choice:

E

Explanation:

Choice E is the correct answer.

The port number 1099 is used for the JRMP naming service and not for object servers. Instead, the ports for object servers are dynamically assigned as in IIOP. Hence, choice E is correct.

The ports 80 and 8080 are typically used for HTTP communication, and the port 443 is used for HTTPS communication. Therefore, the remaining choices are incorrect.

Section 8. Applicability of J2EE technology

Introduction

J2EE is a set of coordinated specifications and practices that together enable solutions for developing, deploying, and managing multi-tier, server-centric applications. J2EE extends the strengths of the Java 2 Platform, Standard Edition (J2SE) to the enterprise level. As you have already seen, EJB is a component architecture model prescribed by J2EE technology for the development of component-based distributed applications. In this section, we look at the application aspects best supported by J2EE and EJB technologies and try to understand the best technology to use for a given scenario.

When to use J2EE?

The cost and complexity of developing and deploying multi-tier solutions are reduced by the J2EE application model because the J2EE container provides a complete set of services to application components and handles many low-level concerns automatically without complex programming. That functionality results in the development of rapid solutions for complex enterprise business problems.

If you have the following requirements, then J2EE is the platform of choice:

- **Faster solutions delivery to market:** J2EE containers separate the business concerns from the system concerns, such as resource management and lifecycle management of components. This means that the programmers can use their time to focus on business solutions rather than on programming low-level system tasks.
- **Freedom of choice:** J2EE can run in a heterogeneous environment and provides complete freedom from vendor lock-in. So, you can deploy an application on an expensive commercial server running on a proprietary OS, as well as on an open source server running on a personal desktop

system, depending on your business needs.

- **Simplified connectivity to legacy systems:** J2EE technology makes it easier to connect the applications and systems already present in the business environment and extends them to new delivery channels such as the Web, cell phones, and smart devices.
- **Investment protection:**The J2EE platform keeps in pace with the technical advancements in the industry. For example, when Web services gained momentum, J2EE adopted it in the immediate revision. This ensures that J2EE technology is future-proof and your IT investment is protected from becoming obsolete.

When to use EJB

As you've already seen, an Enterprise JavaBean is a server-side component that models an application's business logic. An EJB-based solution is not the preferred solution for every kind of problem. Because it is a heavyweight model, it often has a trade-off in terms of performance and complexity. Therefore, using it in a place where it is not required can lead to problems. If your application has the following requirements, then EJB components provide a better choice for implementing your solution.

- **Scalability:** EJB technology is a distributed component technology. As the user base increases, the application components can be distributed across multiple machines. Even though the EJB components run on different machines, the client is completely transparent of the location.
- **Transactionality:** EJB technology supports transactions in both programmatic and declarative ways. Most of the complex processes in transaction handling are automatically managed by the EJB container.
- **Support for a variety of clients:** EJB technology supports various types of clients, such as remote clients, thin clients, and applets.
- **Increased productivity:** Although EJB implementation has a steep learning curve, it pays off eventually because programmers are freed from complex, low-level tasks, such as transactions, persistence, security, and pooling. They can focus on developing the business logic and leave the rest to the application server vendors.
- **Fine-grained security:** EJB has built-in support for both declarative and programmatic role-based security.
- **Increased code reuse:** EJB takes code reuse to a new level. You can write and declaratively customize generic components across various projects simply by configuring the deployment descriptors at deploy time.
- **Need to avoid vendor lock-in:** EJB components extend Sun's write-once-run-anywhere and write-once-deploy-anywhere principles. You

can write components that are portable not just across JVMs but also across application server vendors.

J2EE technologies and their applications

The exam requires you to pick the right J2EE technology for a given problem. The following table lists the various J2EE standard services and the corresponding application aspects they solve.

J2EE services	Application aspect
EJB	• Distributed component programming model • Support for persistence, transactions, and security
JSP	• Handling and processing of HTTP requests and responses • For generating dynamic content in text-based markup languages, such as HTML, XML ,WML, and SVG
Servlets	• Handling and processing of HTTP requests and responses • For handling nontextual data and dispatching requests
JMS	• For communicating with message-oriented middleware (MOM) products in a generic way • For loosely coupled communication • For asynchronous and reliable communication across enterprise components and legacy systems
JDBC	• For accessing data stores in a generic way
JNDI	• For accessing directory and naming services in a generic way
JAXP	• For parsing and transforming XML documents in a generic way
RMI-IIOP	• Simplicity of Java language and interoperability with CORBA is

	required
Java IDL	• For invoking external CORBA objects using IIOP
JTA/JTS	• Demarcating transactions in a generic manner independent of the transaction manager implementation
JavaMail	• Platform-independent and protocol-independent framework to build mail and messaging applications
JCA	• For developing pluggable resource adapters that support access to EIS

Summary

In this section, we discussed how the J2EE platform is suitable when you require faster time-to-market and need to avoid vendor lock-in. You learned that you can choose the EJB component model when you require a robust, scalable, and transactional component model. We also listed various application aspects and the J2EE technologies best suited for solving them. In the exam, you will be given a scenario and asked to choose the appropriate technology for implementing it. Read all the requirements carefully and determine the best possible solution.

Test yourself

Question 1:

Your company has an existing intranet portal developed using CGI and Perl technology. The portal has dynamic pages that fetch data from a relational database for display purposes. Which J2EE technology is best suited for replacing this legacy application?

Choices:

- **A.** JSP/servlets
- **B.** EJB with JSP/servlets
- **C.** JDBC with JSP/servlets
- **D.** JNDI with JSP/servlets

Correct choice:

C

Explanation:

Choice C is the correct answer.

At a cursory glance, choice A might seem like the correct choice. But the portal has to access data from the database for which either EJB or JDBC can be chosen. As per the requirement, the data is read only for display purposes, which means that there isn't any need for transactional support. Hence, choice C is correct while choices A and B are incorrect.

JNDI is used for generic access to directory and naming services. Because there is no such requirement here, choice D is incorrect.

Section 9. Design patterns

Introduction

The concept of design patterns was first introduced by Christopher Alexander when he discovered that you can repeatedly apply certain solutions to solve the same or similar problems. Although his ideas were based on building designs, the concept applies equally well to the software field. Understanding this, the Gang of Four (Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson) published their famous book *Design Patterns: Elements of Reusable Object-Oriented Software*, which earmarked a new era in the software industry. The SCEA exam tests your knowledge of the 23 patterns documented under three different categories by the Gang of Four.

In this tutorial, for each pattern, you will find a catalog containing the following elements.

- Intent: the objective of this pattern
- Description: a brief description of the pattern
- UML diagram: the structure of the pattern documented in UML
- Benefits: the pros of using this pattern
- When to use: the possible scenarios in which you can use the pattern

Benefits of design patterns

Design pattern benefits include:

- **Proven solutions:** Patterns capture the experience and knowledge of developers who have successfully used these patterns in their own work.
- **Promotes reusability:** Patterns provide a ready-made solution you can adapt to different problems, per requirements.
- **Expressive:** Patterns provide a common vocabulary of solutions that can express large solutions in a precise way.

GOF patterns: Creational

Creational patterns specialize in abstracting the instantiation process. They help in isolating how objects are created, composed, and represented from the rest of the system. There are five patterns defined in this category:

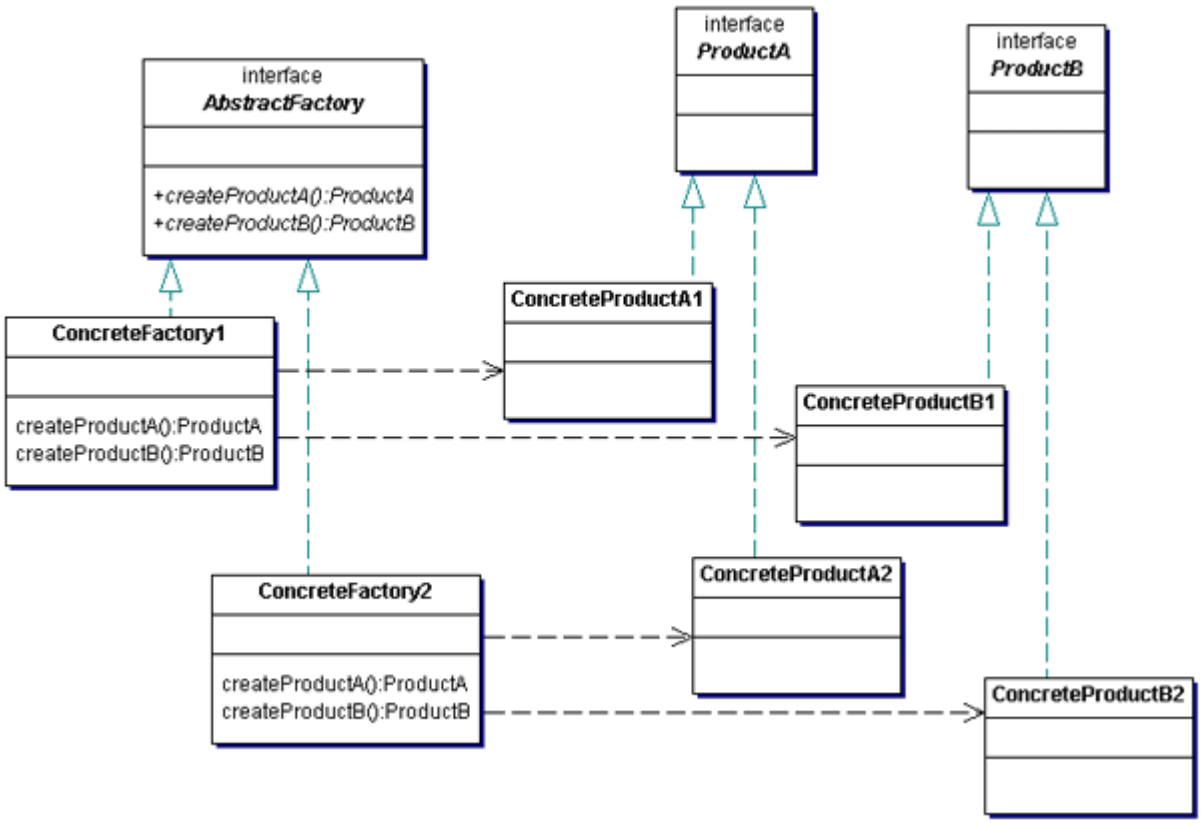
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Abstract Factory

Intent:
Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Description:
The Abstract Factory pattern defines an abstract class that determines the appropriate concrete class to instantiate to create a set of concrete classes that implement a standard interface. The client interacts only with the interfaces and the Abstract Factory class. The client is completely shielded from the concrete classes. The Abstract Factory pattern is similar to the Factory Method pattern, except that it creates families of related objects and can be considered as a factory of factories.

UML diagram:



Benefits:

- Isolates concrete classes.
- Makes exchanging product families easy.
- Promotes consistency among products.

When to use?
You can use the Abstract Factory pattern when:

- The system should be independent of how its products are created, composed, and represented.
- The system should be configured with one of the multiple families of products.
- A family of related product objects is designed to be used together and you need to enforce this constraint.
- You want to provide a class library of products and you want to reveal just their interfaces, not their implementations.

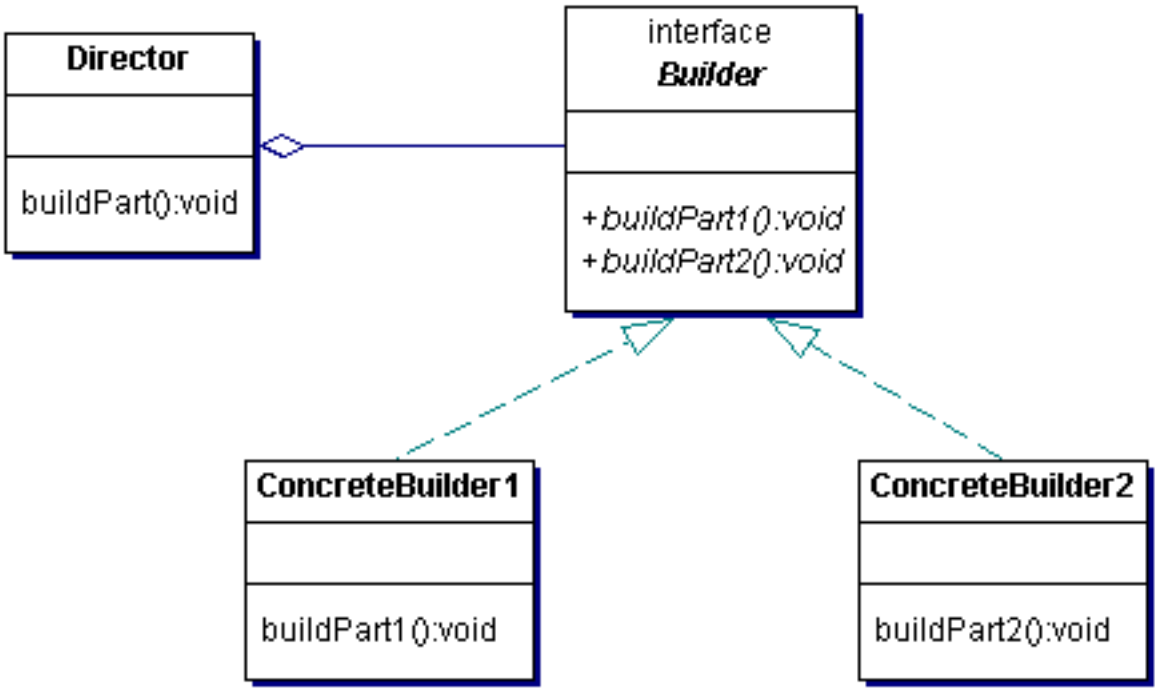
GOF patterns: Creational, continued

Builder

Intent:
Separates the construction of a complex object from its representation so that the same construction process can create different representations.

Description:
The Builder pattern separates the construction of a complex object from its representation so the same construction process can create different objects. The Builder pattern allows a client object to construct a complex object by specifying only its type and content. Based on the type, the appropriate concrete builder takes the responsibility of creating and assembling the complex object. The client is isolated from the details of the object's construction.

UML diagram:



Benefits:

- Lets you vary a product's internal representation.
- Isolates code for construction and representation.
- Gives finer control over the construction process.

When to use?
You can use the Builder pattern when:

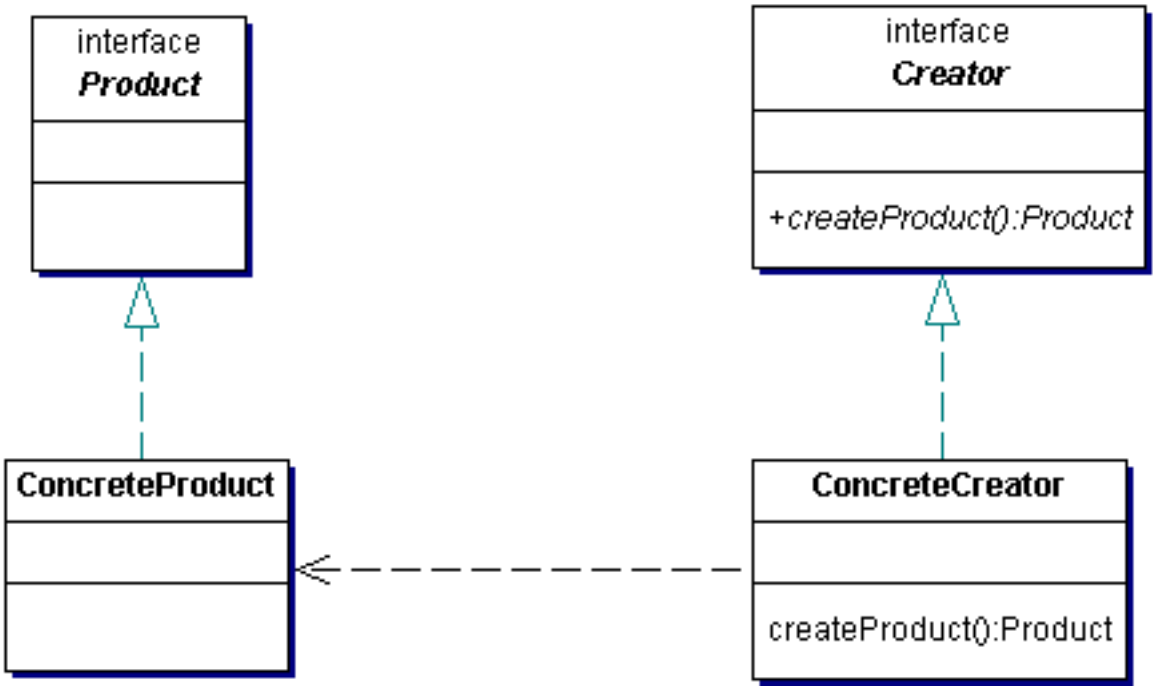
- The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
- The construction process must allow different representations for the constructed object.

Factory Method

Intent:
Defines an interface for creating an object but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Description:
The Factory method lets a class defer instantiation to subclasses, which is useful for constructing individual objects for a specific purpose without the requestor knowing the specific class being instantiated. This helps in introducing new classes without modifying the existing code because the new class implements only the interface, so the client can use it. You create a new factory class to create the new class, and the factory class implements the factory interface.

UML diagram:



Benefits:

- Eliminates the need to bind application-specific classes into your code.
- Gives subclasses a hook for providing an extended version of an object.
- Connects parallel class hierarchies and localizes the knowledge about dependency between the classes.

When to use?
You can use the Factory Method pattern when:

- A class cannot anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.

- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

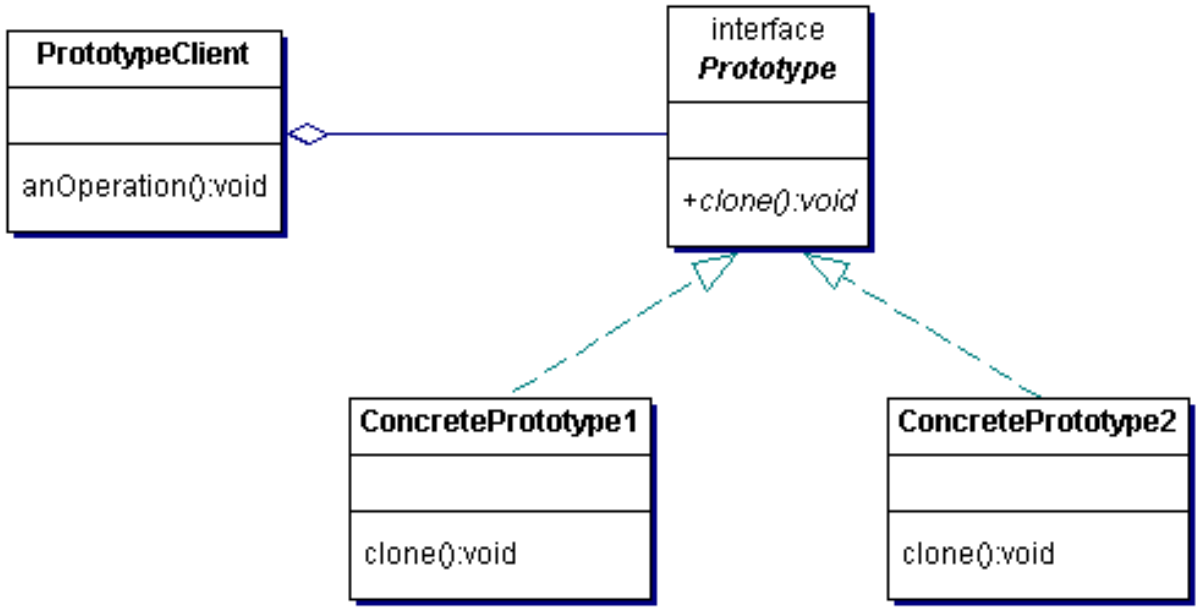
GOF patterns: Creational, continued

Prototype

Intent:
Specifies the kinds of objects to create using a prototypical instance and creates new objects by copying this prototype.

Description:
The Prototype pattern allows an object to create customized objects without knowing their exact class or the details of how to create them. The Prototype pattern gives prototypical objects to an object and then initiates the creation of objects. The creation-initiating object then creates objects by asking the prototypical objects to make copies of them.

UML diagram:



Benefits:

- Hides the concrete product classes from the client.
- Adding and removing products at runtime.
- Specifying new objects by varying values.
- Specifying new objects by varying structure.
- Reduced subclassing.

- Configuring an application with classes dynamically.

When to use?
You can use the Prototype pattern when:

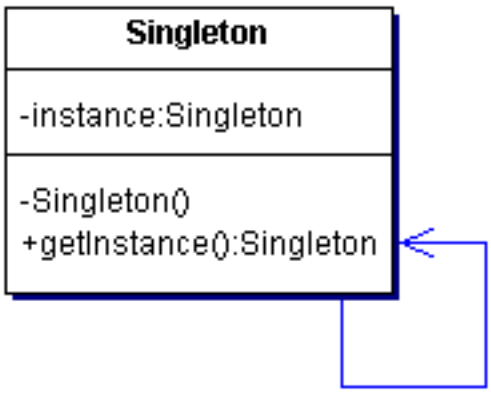
- A system should be independent of how its products are created, composed, and represented.
- The classes to instantiate are specified at runtime, for example, by dynamic loading.
- You have to avoid building a class hierarchy of factories that parallels the class hierarchy of products.
- Instances of a class can have one of only a few different combinations of state.

Singleton

Intent:
Ensures that a class has only one instance and provides a global point of access to it.

Description:
The Singleton pattern ensures that a class has only one instance and provides a global point of access to that class. Generally, you implement this by hiding the constructor and providing a method that ensures that all the objects that require an instance of this class use the same instance.

UML diagram:



Benefits:

- Controlled access to sole instance.
- Reduced namespace.
- Permits refinement of operations and representation.
- Permits a variable number of instances.
- More flexible than class operations.

When to use?
You must use the Singleton pattern when:

- There must be exactly one instance of a class and it must be accessible to clients from a well-known access point.

GOF patterns: Behavioral

Behavioral patterns are concerned with algorithms and the assignment of responsibilities to objects. They document the patterns of objects as well as the patterns of communication between them. There are 11 patterns defined in this category:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

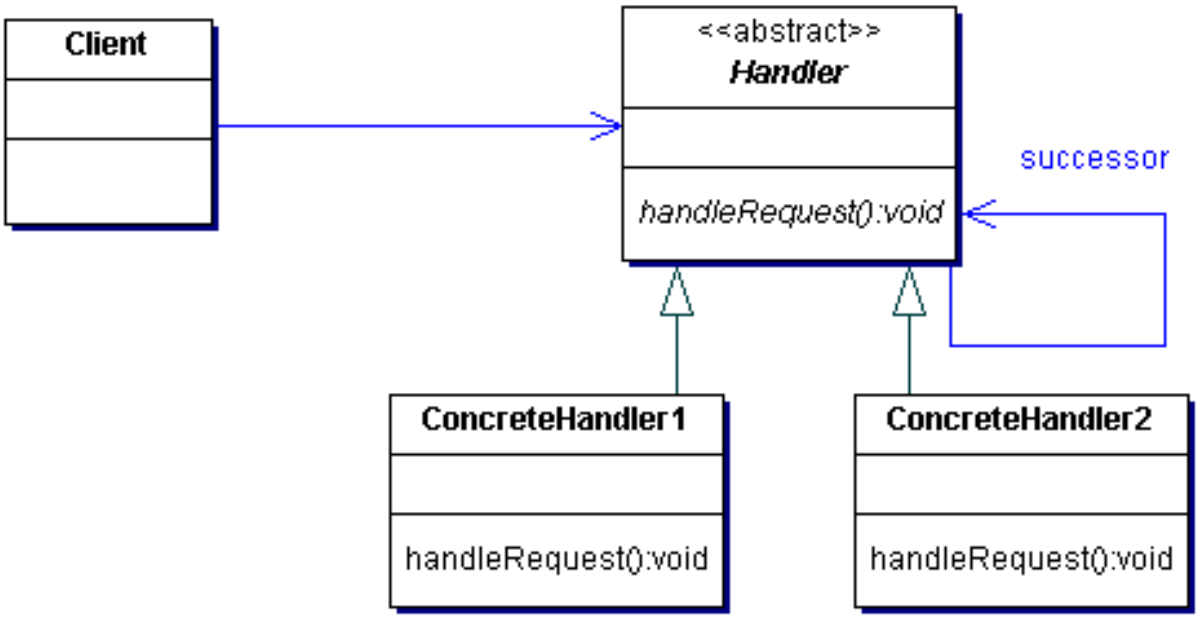
Chain of Responsibility

Intent:
Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chains the receiving objects and passes the request along the chain until an object handles it.

Description:

This pattern decouples senders and receivers by giving multiple objects a chance to handle a request. The request gets passed along a chain of objects until one of them handles it.

UML diagram:



Benefits:

- Reduced coupling.
- Added flexibility in assigning responsibilities to objects.

When to use?
You can use the Chain of Responsibility pattern when:

- More than one object can handle a request, and the handler isn't known.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

GOF patterns: Behavioral, continued

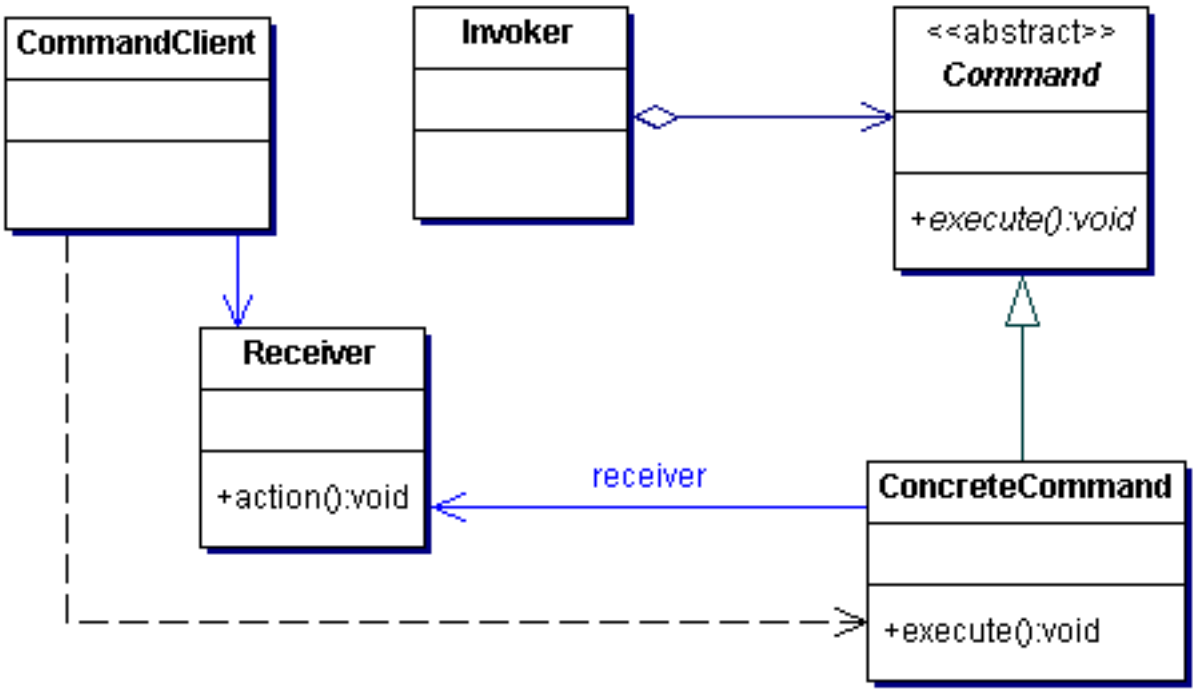
Command

Intent:
Encapsulates a request as an object, thereby letting you parameterize clients with different requests (queue or log requests) and support undoable operations.

Description:

Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. The Command pattern encapsulates a request in an object, which allows storing the command, passing the command to a method, and returning the command like any other object.

UML diagram:



Benefits:

- Decouples the object that invokes the operation from the one that knows how to perform it.
- Easy to add new commands because you don't have to change existing classes.

When to use?
You can use the Command pattern when:

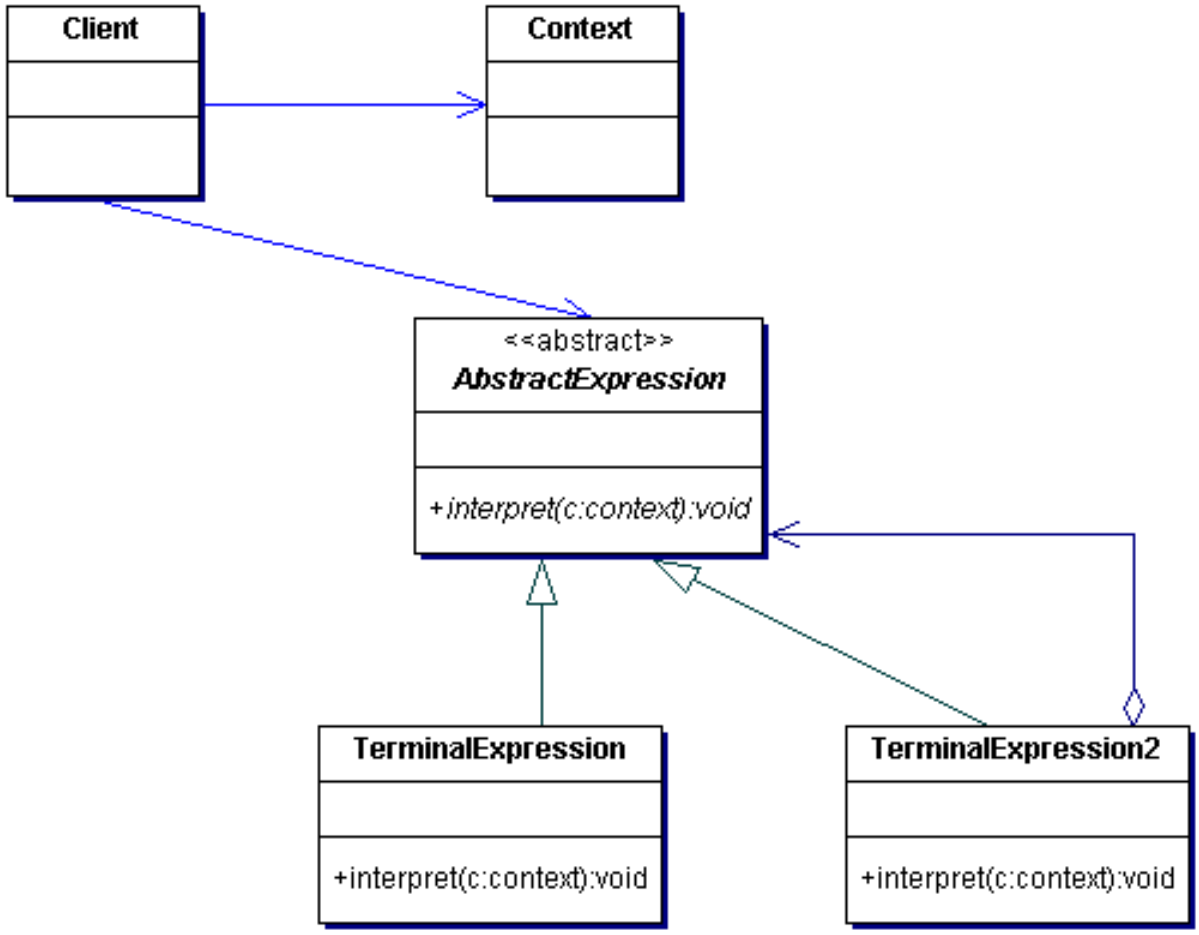
- You want to parameterize objects by an action to perform.
- You specify, queue, and execute requests at different times.
- You must support undo, logging, or transactions.

Interpreter

Intent:
Given a language, defines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Description:
If a particular kind of problem occurs often enough, it might be better to express instances of the problem as sentences in a simple language. You can then build an interpreter that solves the problem by interpreting these sentences. The Interpreter pattern describes how to define the grammar for such simple languages, represent sentences in the language, and interpret these sentences. It uses a class to represent each grammar rule.

UML diagram:



Benefits:

- It is easy to change and extend the grammar.
- Implementing the grammar is simple.
- Adds new ways to interpret expressions.

When to use?
You can use the Interpreter pattern when:

- The grammar of the language is simple.
- Efficiency is not a critical concern.

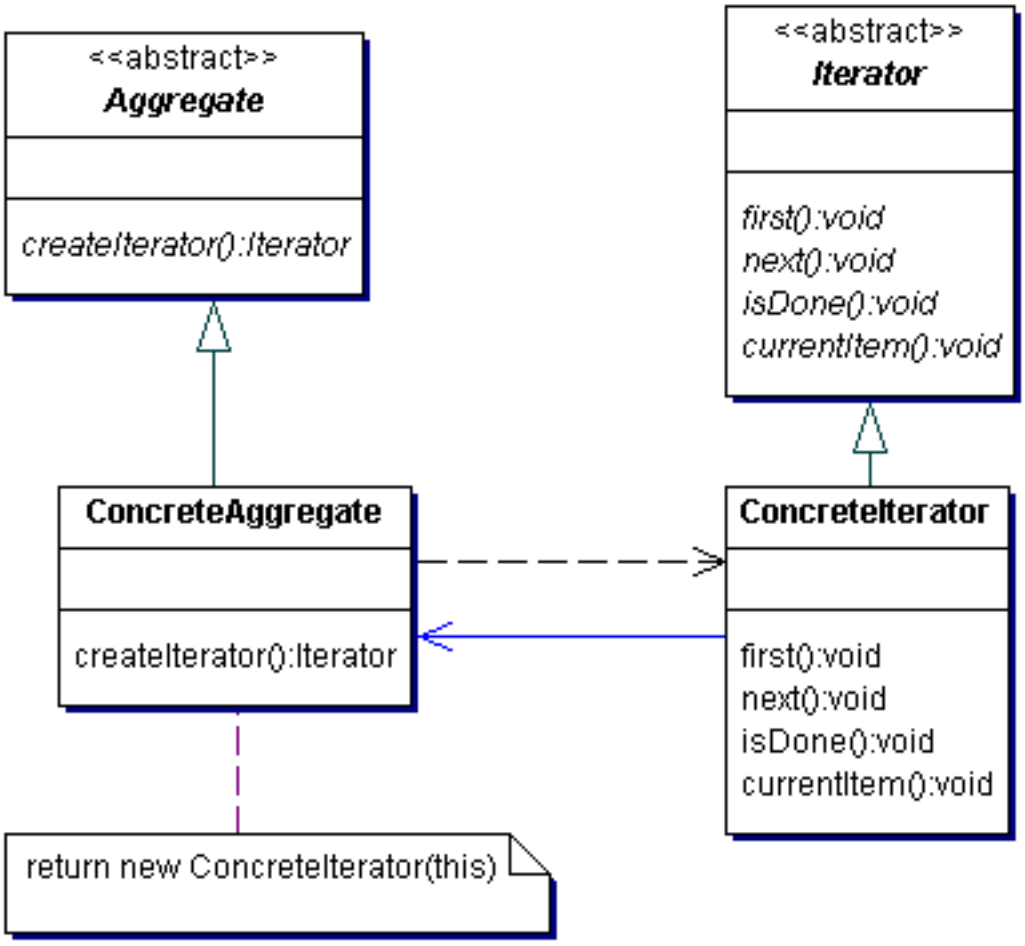
GOF patterns: Behavioral, continued

Iterator

Intent:
Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Description:
The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into a separate iterator object responsible for tracking the current element; that is, it knows which elements have been traversed already.

UML diagram:



Benefits:

- Supports variations in the traversal of an aggregate.
- Simplifies the Aggregate interface.

When to use?

You can use the Iterator pattern when you want to:

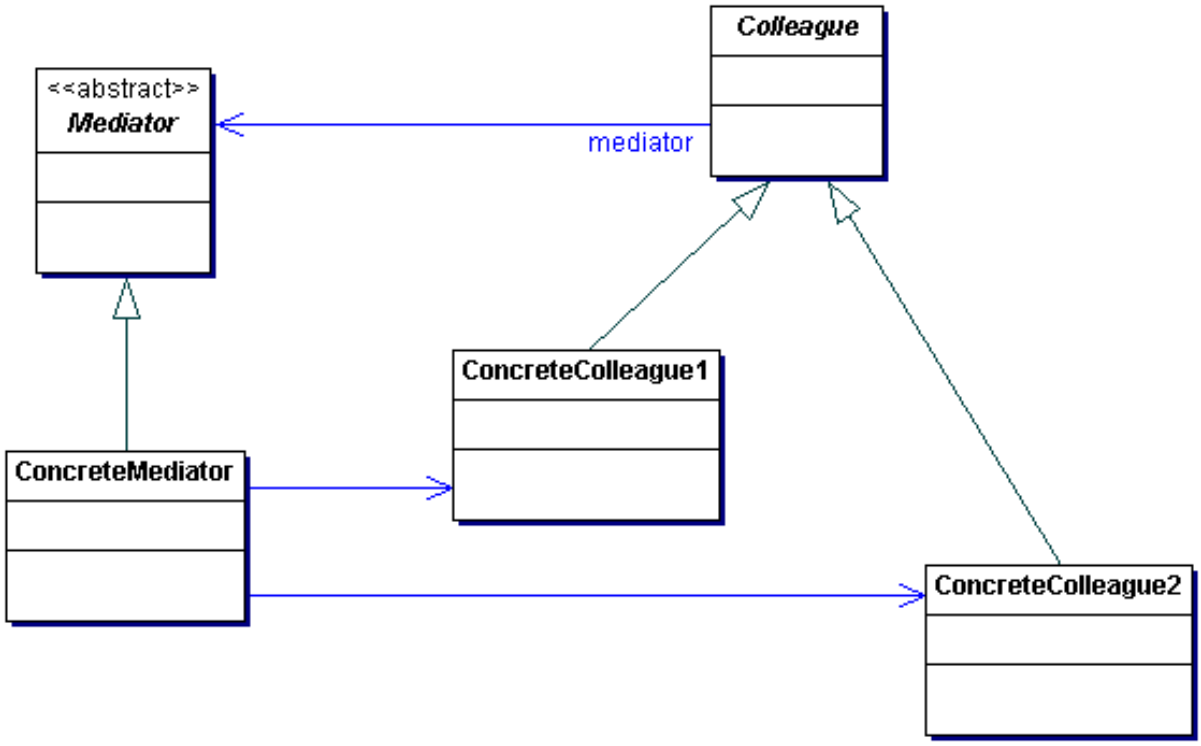
- Access a collection object's contents without exposing its internal representation.
- Support multiple traversals of objects in a collection.
- Provide a uniform interface for traversing different structures in a collection.

Mediator

Intent:
Defines an object that encapsulates how a set of objects interacts. Mediator promotes loose coupling by keeping objects from referring to each other explicitly and lets you vary their interaction independently.

Description:
Object-oriented design encourages the distribution of behavior among objects. Such distribution can result in an object structure with many connections between objects; in the worst case, every object has knowledge of the other. You can avoid that by encapsulating collective behavior in a separate mediator object. A mediator is responsible for controlling and coordinating the interactions of a group of objects. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator, thereby reducing the number of interconnections.

UML diagram:



Benefits:

- Decouples colleagues.
- Simplifies object protocols.
- Abstracts how objects cooperate.
- Centralizes control.

When to use?
You can use the Mediator pattern when:

- A set of objects communicates in well-defined but complex ways.
- Reusing an object is difficult because it refers to and communicates with many other objects.
- A behavior distributed between several classes should be customizable without introducing a lot of subclasses.

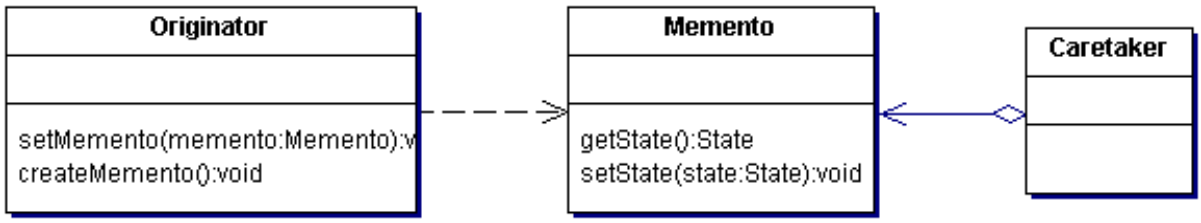
GOF patterns: Behavioral, continued

Memento

Intent:
Without violating encapsulation, the Memento pattern captures and externalizes an object's internal state so that the object can be restored to this state later.

Description:
A memento is an object that stores a snapshot of the internal state of another object -- the memento's originator. The originator initializes the memento with information that characterizes its current state. Only the originator can store and retrieve information from the memento; the memento is "opaque" to other objects.

UML diagram:



Benefits:

- Preserves encapsulation boundaries.
- Simplifies the originator.

When to use?
You can use the Memento pattern when:

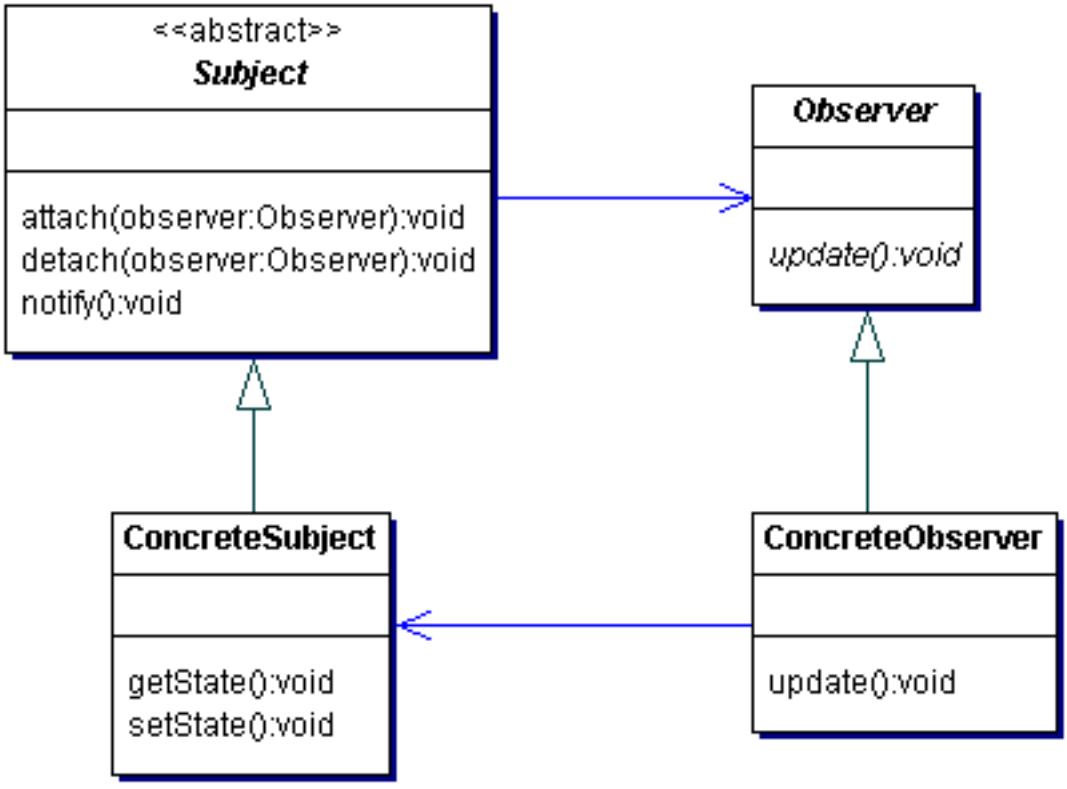
- A snapshot of an object's state must be saved so it can be restored to that state later.
- A direct interface to obtain the state would expose implementation details and break the object's encapsulation.

Observer

Intent:
Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are automatically notified and updated.

Description:
The Observer pattern provides a way for a component to flexibly broadcast messages to interested receivers. The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer queries the subject to synchronize its state with the subject's state.

UML diagram:



Benefits:

- Abstract coupling between subject and observer.
- Support for broadcast communication.

When to use?
You can use the Observer pattern when:

- An abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- A change to one object requires changing others, and you don't know how many objects need to be changed.
- An object should be able to notify other objects without making assumptions about who these objects are.

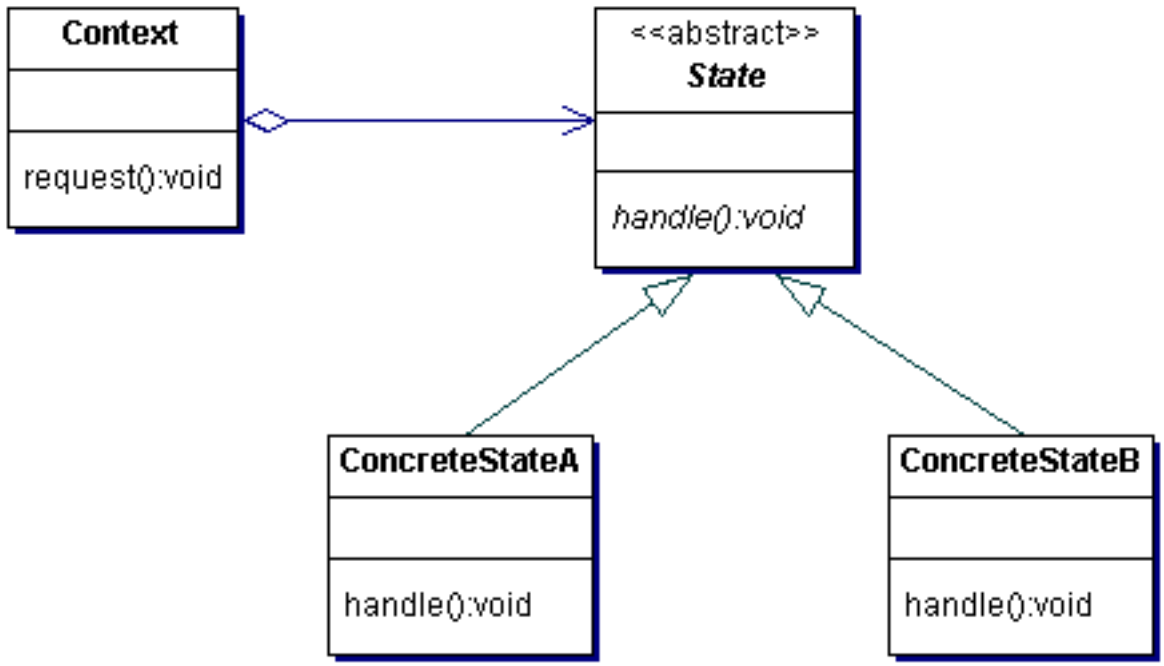
GOF patterns: Behavioral, continued

State

Intent:
Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Description:
The key idea in this pattern is to introduce an abstract class to represent the possible states of the object. This class declares an interface common to all the classes that represent different operational states. The concrete subclasses implement state-specific behavior. Based on the current state, the appropriate concrete class is selected and used.

UML diagram:



Benefits:

- Localizes state-specific behavior and partitions behavior for different states.
- Makes state transitions explicit.

When to use?
You can use the State pattern when:

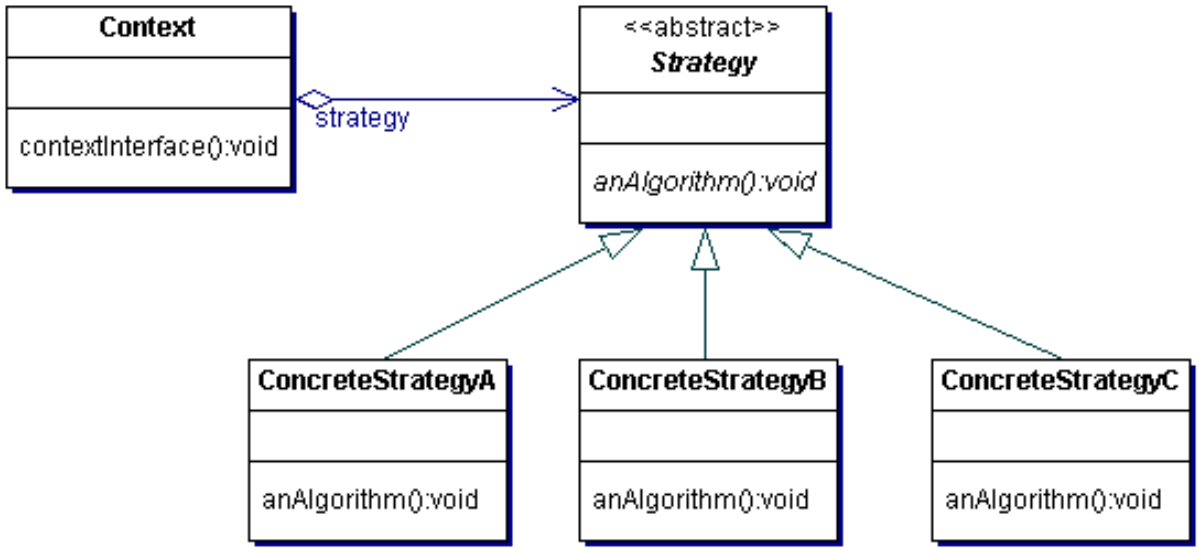
- An object's behavior depends on its state and it must change its behavior at runtime depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state.

Strategy

Intent:
Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Description:
The Strategy pattern defines a group of classes that represent a set of possible behaviors. The functionality differs depending on the strategy (algorithm) chosen.

UML diagram:



Benefits:

- An alternative to subclassing.
- Strategies eliminate conditional statements.
- A choice of implementations.

When to use?
You can use the Strategy pattern when:

- Many related classes differ only in their behavior.
- You need different variants of an algorithm.
- An algorithm uses data that clients shouldn't know about.
- A class defines many behaviors, and these behaviors appear as multiple conditional statements in its operations. Instead of having many conditionals, move related conditional branches into their own Strategy class.

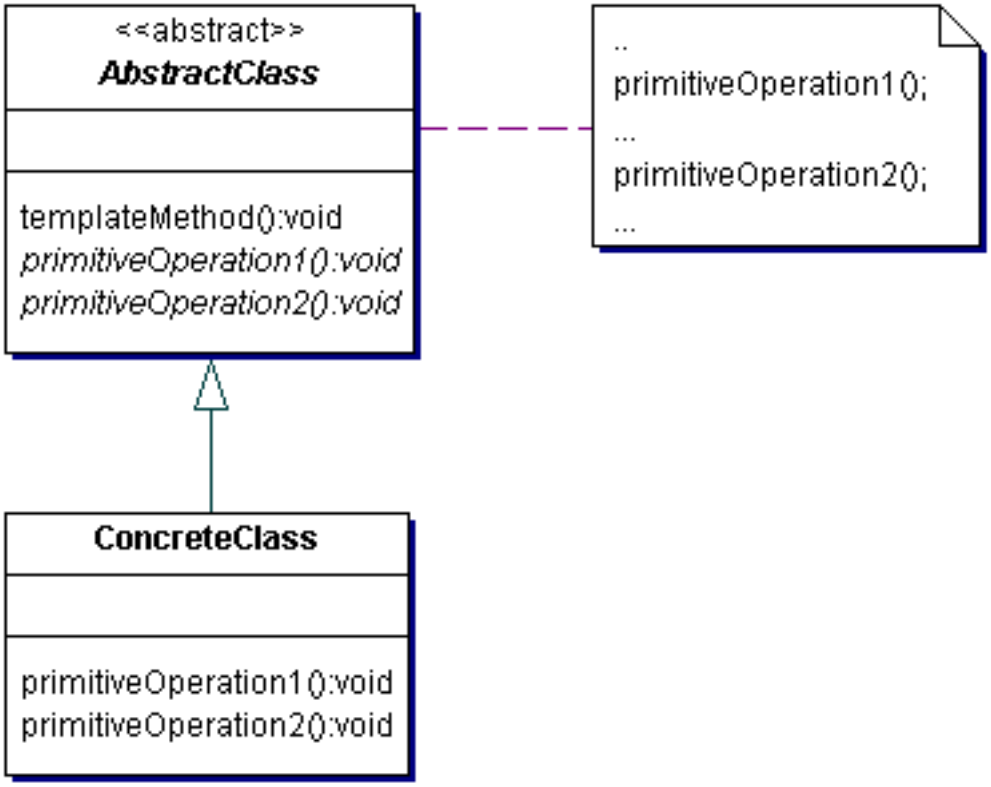
GOF patterns: Behavioral, continued

Template Method

Intent:
Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. The Template Method pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Description:
A template method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior.

UML diagram:



Benefit:

- Template methods provide a fundamental technique for code reuse.

When to use?
You can use the Template Method pattern:

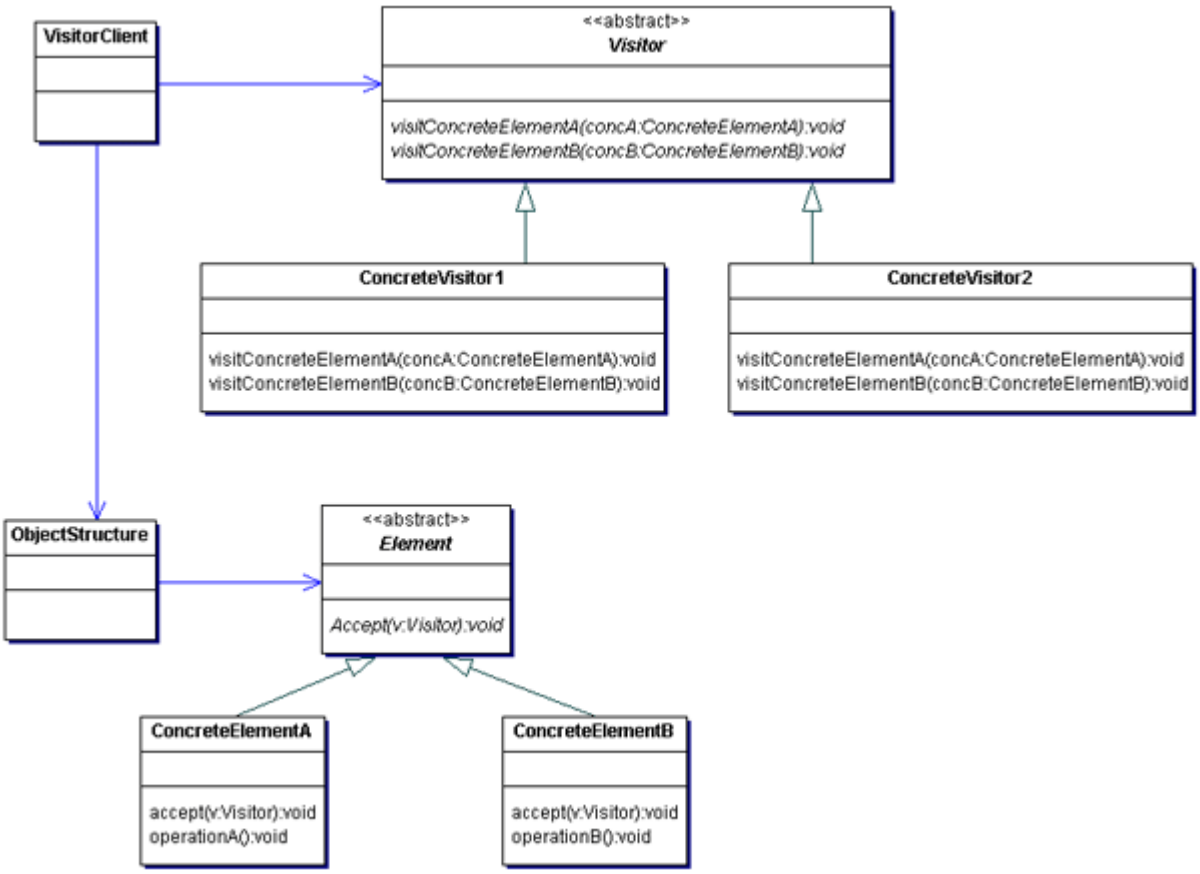
- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
- To control subclasses extensions.

Visitor

Intent:
Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Description:
The Visitor pattern provides a maintainable, easy way to represent an operation to be performed on the elements of an object structure. New operations can be defined without changing the classes of the elements on which it operates.

UML diagram:



Benefits:

- Makes adding new operations easy.
- Gathers related operations and separates unrelated ones.

When to use?
You can use the Visitor pattern when:

- An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure.

GOF patterns: Structural

Structural patterns are concerned with how classes and objects are composed to

form larger structures. Seven patterns are defined in this category:

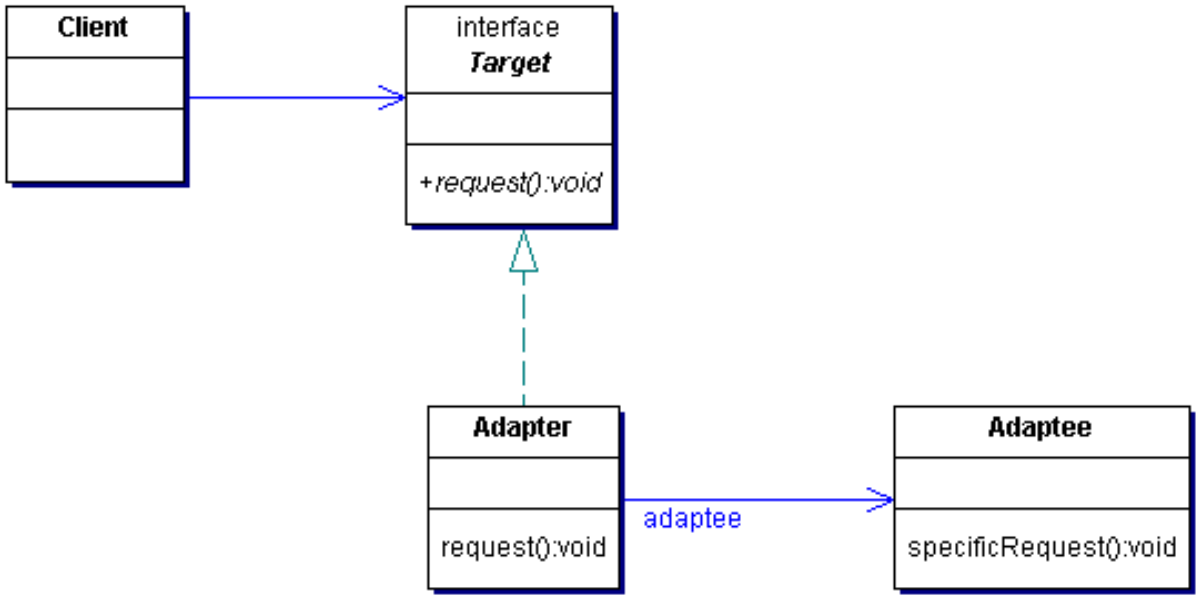
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Adapter

Intent:
Converts the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't, otherwise, because of incompatible interfaces.

Description:
The Adapter pattern defines an intermediary between two classes, converting the interface of one class so it can be used with the other. This enables classes with incompatible interfaces to work together. The Adapter class implements an interface used by the clients and provides access to an instance of a class not known to its clients.

UML diagram:



Benefits:

- Allows two or more incompatible objects to communicate and interact.

- Improves reusability of older functionality.

When to use?
You can use the Adapter pattern when:

- You want to use an existing class and its interface does not match the one you need.
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- You want to use an object in an environment that expects an interface different from the object's interface.

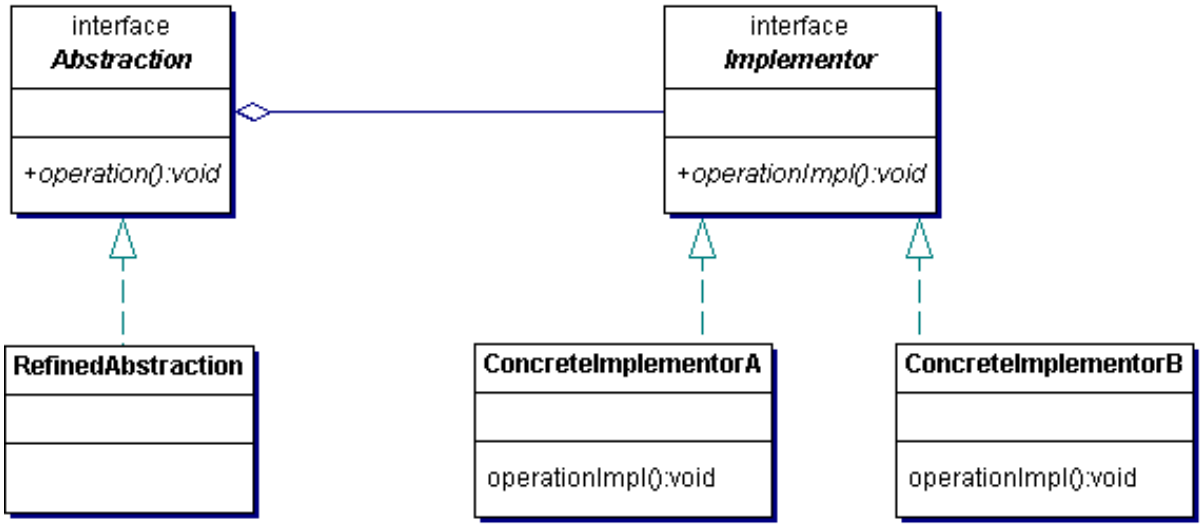
GOF patterns: Structural, continued

Bridge

Intent:
Decouples an abstraction from its implementation so the two can vary independently.

Description:
When an abstraction can have one of the several possible implementations, the usual way to accommodate them is to use inheritance that binds an implementation to the abstraction permanently, making it difficult to modify, extend, and reuse abstractions and implementations independently. Instead of combining the abstractions and implementations into many distinct classes, the Bridge pattern implements the abstractions and implementations as independent classes that can be dynamically combined.

UML diagram:



Benefits:

- Decouples interface and implementation.
- Improves extensibility.
- Hides implementation details from clients.

When to use?
You should use the Bridge pattern when:

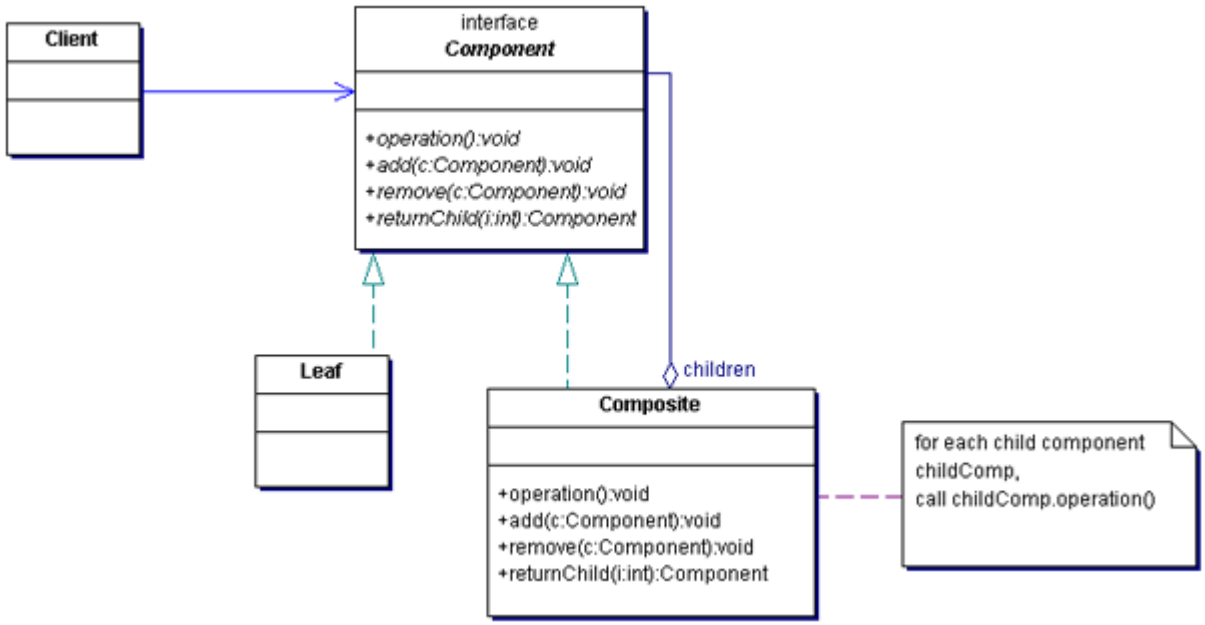
- You want to avoid a permanent binding between an abstraction and its implementation.
- Both the abstractions and their implementations should be extensible using subclasses.
- Changes in the implementation of an abstraction should have no impact on clients; that is, you should not have to recompile their code.

Composite

Intent:
Composes objects into tree structures to represent part-whole hierarchies. The Composite pattern lets clients treat individual objects and compositions of objects uniformly.

Description:
Composite pattern combines objects into tree structures to represent either the whole hierarchy or a part of the hierarchy. This recursive composition allows clients to treat individual objects and compositions of objects uniformly.

UML diagram:



Benefits:

- Defines class hierarchies consisting of primitive objects and composite objects.
- Makes the client simple.
- Makes it easier to add new kinds of components.

When to use?
You should use the Composite pattern when:

- You want to represent part-whole hierarchies of objects.
- You want clients to be able to ignore the difference between compositions of objects and individual objects.

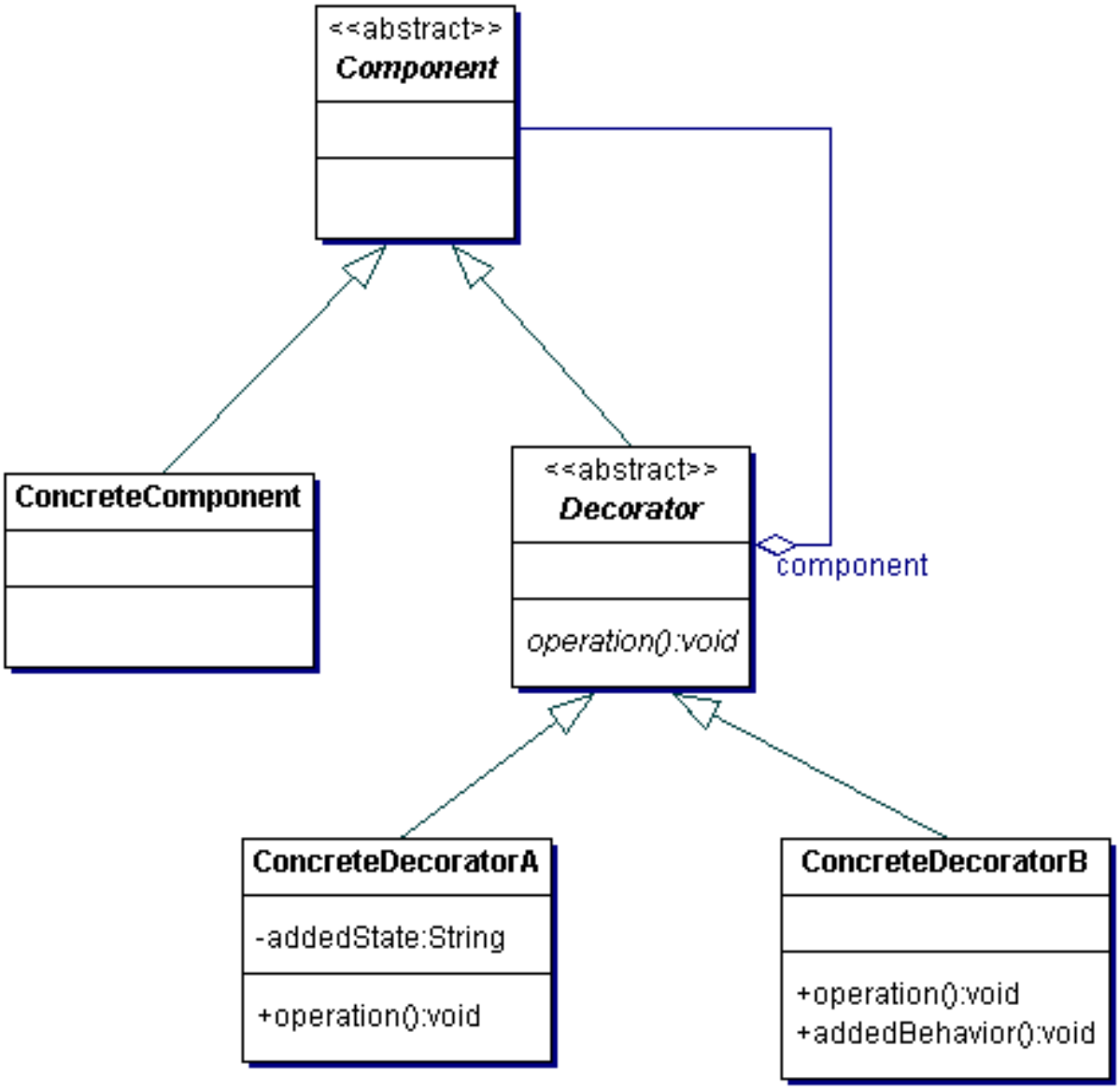
GOF patterns: Structural, continued

Decorator

Intent:
Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Description:
You can use inheritance to add more responsibility to a class. A more flexible approach is to enclose the component in another object that adds the responsibility. The enclosing object is called a decorator. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and might perform additional actions before or after forwarding. The decorators can be nested recursively, thereby allowing an unlimited number of added responsibilities.

UML diagram:



Benefits:

- More flexibility than static inheritance.
- Avoids feature-rich classes high up in the hierarchy because you can define a simple base class and add functionality incrementally as you require decorator objects.

When to use?
You can use the Decorator pattern when:

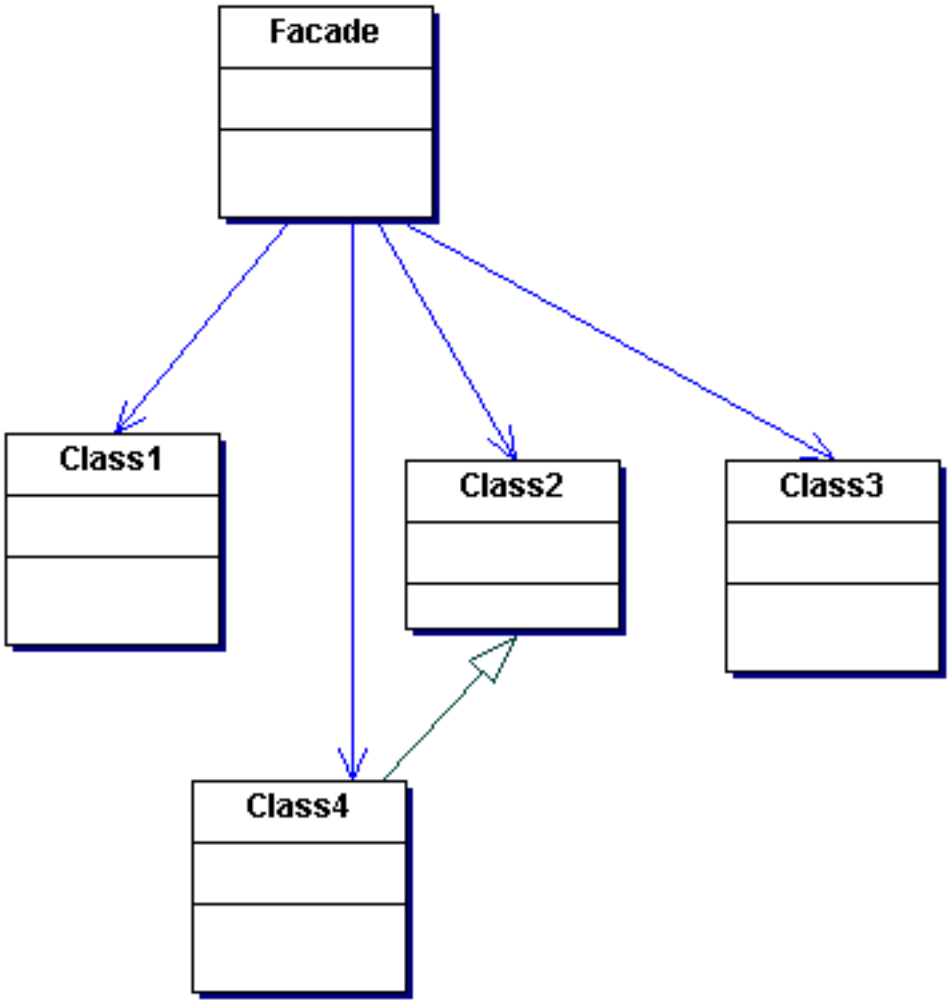
- You want to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- You want to add responsibilities that can be withdrawn later.
- When extension by subclassing is impractical.

Facade

Intent:
Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher level interface that makes the subsystem easier to use.

Description:
The Facade pattern defines a higher level interface that makes the subsystem easier to use because you only have to deal with one interface. This unified interface enables an object to access the subsystem using the interface to communicate with the subsystem.

UML diagram:



Benefits:

- Provides a simple interface to a complex system without reducing the options provided by the system.
- Shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier

to use.

- Promotes weak coupling between the subsystem and its clients.

When to use?
You can use the Facade pattern when:

- You want to provide a simple interface to a complex subsystem.
- There are many dependencies between clients and the implementation classes of an abstraction.
- You want to layer your subsystems.

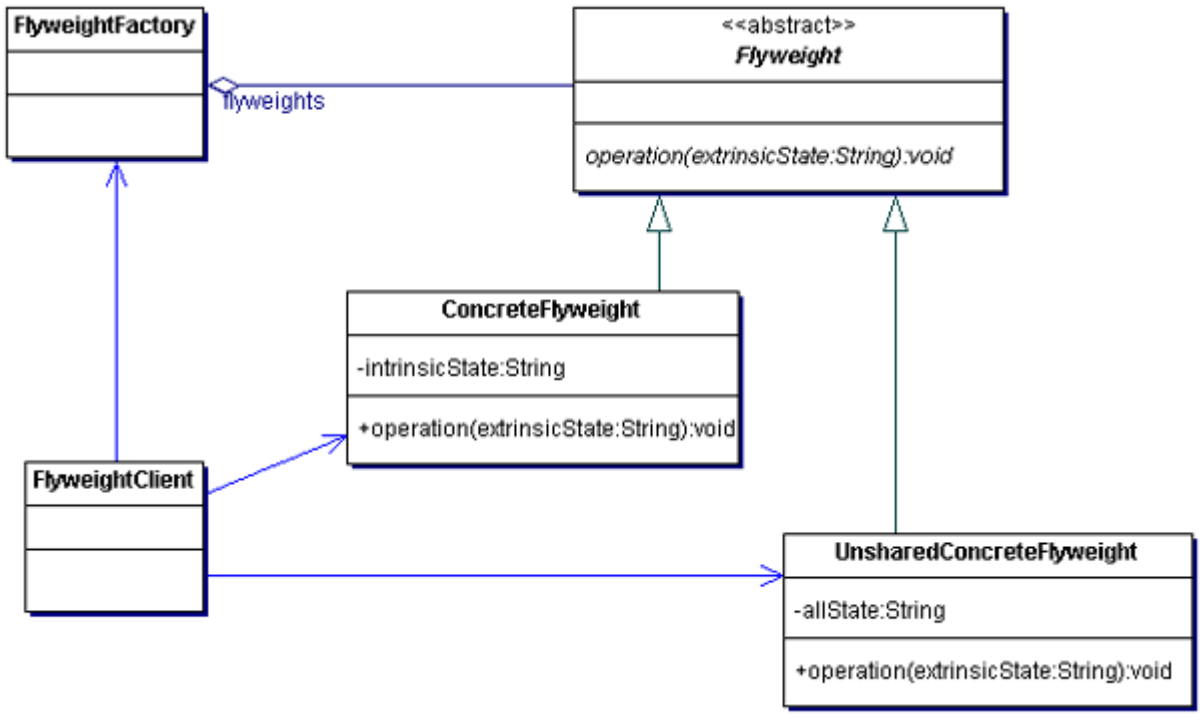
GOF patterns: Structural, continued

Flyweight

Intent:
Uses sharing to support a large number of fine-grained objects efficiently.

Description:
The Flyweight pattern uses a shared flyweight object that can be used in multiple contexts simultaneously. The flyweight has a shareable intrinsic state consisting of information independent of the flyweight's context and a nonshareable extrinsic state that depends on, and varies with, the flyweight's context. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

UML diagram:



Benefits:

- Reduces the number of objects to handle.
- Reduces the memory and storage requirements.

When to use?
You can use the Flyweight pattern when:

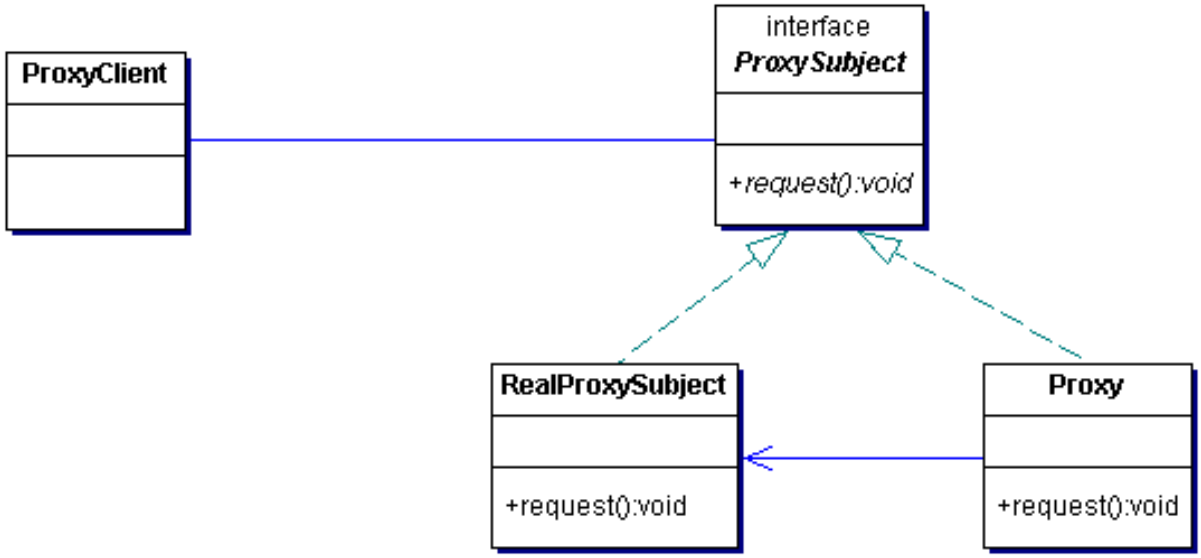
- An application uses a large number of objects.
- Storage costs are high because of the high number of objects.
- Most of the object's state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects, once extrinsic state is removed.
- The application doesn't depend on object identity. Because flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Proxy

Intent:
Provides a surrogate or placeholder for another object to control access to it.

Description:
The Proxy pattern uses a proxy object that acts as a stand-in for a real object. The proxy acts just like the real object and takes care of controlling the access to the real object.

UML diagram:



Benefits:

- A remote proxy can hide the fact that an object resides in a different address space.
- A virtual proxy can perform optimizations, such as creating an object on demand.
- Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

When to use?
You can use the Proxy pattern:

- Whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.

Patterns in J2EE

The following table summarizes the common uses of some of the Gang of Four patterns in the context of J2EE.

Feature	Pattern
EJB Home	• Factory • Abstract Factory
EJB Remote	• Proxy • Facade • Decorator
Session beans that model workflow	• Facade
EJB instance pooling	• Flyweight
Value object	• Memento
JDBC Resultset	• Iterator
JDBC, JMS, JCA, or any bridging technology	• Bridge
ServletContext	• Singleton
Servlet filters	• Chain of Responsibility

	• Decorator
JMS	• Observer • Mediator

Summary

In this section, you were introduced to design patterns and their benefits followed by the brief treatment of the 23 patterns documented by the Gang of Four. As the intent of each pattern captures the problem each pattern tries to solve, it is important to remember and understand the intentions and problems well. The knowledge of the UML structure, benefits, and applicable scenarios are equally important to score well in this section.

Test yourself

Question 1:

You are writing a common transaction framework to manage transactions for all the other subsystems within your organization. The users of this framework should be isolated from the complexities of the framework. Which design pattern should you use to achieve this goal?

Choices:

- **A.** Composite
- **B.** Facade
- **C.** Decorator
- **D.** Adapter
- **E.** Mediator

Correct choice:

B

Explanation:

Choice B is the correct answer.

You would use the Facade pattern to provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher level interface that makes the subsystem easier to use. Because you want to isolate the internal complexities of your framework, Facade is the best solution for this problem. Hence, choice B is correct.

Although Adapter is similar to Facade, it does not help in defining a simplified interface. Instead, you would use it to simply reuse the existing interface. Therefore, choice D is incorrect.

None of the other choices helps in hiding the complexity involved in accessing the system so they are incorrect.

Section 10. Messaging

Introduction

A message is a unit of serializable data exchanged between two or more distributed components running in the same machine or different machine. By using a message-oriented middleware (MOM) infrastructure, an application can create, send, and receive messages. This lets you combine separate business components into a reliable, yet flexible system. Several vendors provide MOM, and in J2EE, Java Messaging Service (JMS) offers a generic way to access these systems. In this section, let's explore the different messaging modes and models and learn how to choose the correct one for a given scenario.

Communication modes

There are two modes of communication depending on the level of coupling between the sender and receiver:

- Synchronous
- Asynchronous

Synchronous

A distributed component sends a message to another active component and waits for the reply (also known as blocking call) to proceed further. The synchronous communication is tightly coupled because both the sender and receiver have knowledge about each other. The sender is responsible for retries in case of failures.

The benefits of synchronous communication are:

- This mode is fail-safe and is used for transaction processing.
- Sender can receive the response immediately in realtime.
- When multiple messages are sent, they reach the destination in the same order in which they are sent.

- It is a reliable communication mode.

You can use this mode when the sender:

- Wants to have more control over the message.
- Needs real-time response.
- Wants to maintain the order of message processing.
- Wants to retry in case of message failure.

Asynchronous

A distributed component can send messages to any other component via MOM and continue its processes without waiting for the response. This communication mechanism is loosely coupled, where sender and receiver need not have knowledge about each other because a central intermediary, the MOM, exists. Messages can arrive at the destination in any sequence and not necessarily in same order in which they are sent. MOM is responsible for retry in case of failure in the communication.

The benefits of asynchronous communication are:

- Sender need not wait till the message gets processed. The responsibility is delegated to the MOM.
- Messages can be queued. Sender and receiver need not be always available to receive the messages.
- It is loosely coupled because the sender and receiver do not directly communicate; they communicate through MOM.

You can use this mode when the sender:

- Wants to broadcast the message.
- Needs no response, or the response is not needed immediately.
- Wants to use the system hardware efficiently.
- Wants to do transaction processing in high volume.

Messaging models

There are two types of messaging models depending on whether you require one-to-one message delivery or a one-to-many broadcast delivery. The models are:

- Point-to-point messaging
- Publish-subscribe messaging

Point-to-point messaging

In this model, the sender sends the messages to a destination known as queue, and the receiver consumes the message from the queue. A queue is designated to only one receiver. More than one sender can send the messages to a queue, but only one consumer receives that message from the queue. The messages in the queue are processed on a first-in-first-out (FIFO) basis. The messages stay in the queue until they are consumed by the consumer or until the messages' expiry time. A sender can also send the message directly to the consumer instead of placing it in the queue. A consumer can acknowledge the successful processing of the message.

Publish-subscribe messaging

In this model, publishers publish the messages to a topic. The subscribers who subscribe to that topic then receive the message. The message stays in the topic until it is sent to the active subscribers. If some subscribers are not active, the messages are not delivered to them. There is a special type of subscription known as a durable subscription in which the messages will not be lost if the subscriber is not active. Rather, the messages are delivered once the subscriber becomes active.

Usage scenarios

The following table summarizes the technologies you can use for different scenarios.

Technology	Scenarios
Messaging	• Need to broadcast messages • To interface between two incompatible systems that do not communicate directly • To simulate threads • Asynchronous communication
EJB	• For doing transactional and secure operations • Need an immediate response • To perform business logic • To maintain persistence data
Messaging and EJB	• To retrieve data and send to another system, which does not communicate directly • To perform distributed transactions across multiple application and systems

Summary

In this section, we discussed the different types of communication modes and messaging models used in enterprise systems for messaging. We also discussed their benefits and when to use these different communication modes. In the exam, given a scenario, you should be able to identify which technologies you can use for the particular scenario. Remember that if the scenario calls for one-to-one messaging then it is a candidate for the point-to-point messaging model. On the other hand, if there are numerous receivers, the best method for implementation is the publish-subscribe model.

Test yourself

Question 1:

Which of the following choices describe asynchronous messaging?

Choices:

- **A.** Loose coupling between sender and receiver
- **B.** Blocks until message is processed
- **C.** Suitable for transaction processing
- **D.** The network is not required to be available

Correct choice:

A and **D**

Explanation:

Choices A and D are the correct answers.

Asynchronous messaging is loosely coupled because the sender and receiver do not directly communicate; they use MOM for communication. Hence, choice A is correct.

Because MOM takes care of delivering the messages if the receiver is not available, the network is not required to be constantly running. Thus, choice D is correct.

Only synchronous messaging blocks the sender until the receiver processes the message. Therefore, choice B is incorrect.

Because transaction processing requires immediate response, only synchronous messaging is suitable for implementing it. Hence, choice C is incorrect.

Section 11. Internationalization

Introduction

In this age of the Internet, businesses are no longer confined to the boundaries of a country or culture. Now, it's just as easy to access an application located at the other end of the globe, as it is to access an application in the same building. Due to these advancements, there is an increasing need to develop multilingual and multicultural applications. Developing multiple versions of the same application to cater to different locales is often cumbersome and might lead to maintenance nightmares. An easier option is to use internationalization by designing an application in such a way that it adapts to the preferences of the country where it is used without any change in the source code. The term internationalization is often abbreviated as *i18n* because there are 18 letters between "i" and "n." Making an application adaptable to a country's local preferences is called localization, and the corresponding abbreviation is *l10n*.

Internationalizable elements (need for I18n)

This is a common list of elements that should support internationalization:

- Text output messages
- Unicode
- Security (encryption, decryption) algorithms due to government restrictions
- Dictionary sorting order
- Formats: number, date, currency, time, measurement, postal code
- GUI items: labels, menu, buttons, online help, colors, page layouts
- Sounds and graphics
- Tax and other legal rules
- Cultural preferences

Almost any feature of the program can be localized. With the addition of localized data, the same executable code runs in different geographical locations. GUI components such as labels and menus are not coded in the program and are retrieved from the outside depending upon the location in which they get executed. Internationalized programs need not be recompiled to support any language or location. You can localize the program just by adding the property files that

correspond to the local language.

Java 2 internationalization features

Java programming language has powerful APIs that support internationalization/localization. The APIs used for this purpose are Properties, Locale, ResourceBundle, Unicode, java.text package, InputStreamReader, and OutputStreamWriter.

Properties

The Properties class is used for loading values from property files at startup or runtime. Only string objects can be stored in a property file as key-value pairs. The properties can be loaded from or saved to a stream. Generally, a property file contains data about the application characteristics or its environment. A properties object, by using the `load()` method, can read a localized properties file or any arbitrary input stream to access the appropriate localized values.

```
Properties props = new Properties();
String myProps = "MyProperties";
props.load(new BufferedInputStream(new FileInputStream(myProps);
String value = System.getProperty("key");
```

This code snippet shows how the value is retrieved for the key. If the key is not present, the `getProperty()` method returns null.

Locale

A `java.util.Locale` object is used to identify the specific locale for a particular session or user. A locale object is the identifier of a particular region. You can set the locale object with any value depending on the user's preferences. It is represented in one, two, or three elements. The first part contains the language code; the second part contains the country code; and the third part is the variant. A variant allows more than one locale for a language and country combination. A locale is represented as language, and country code separated by an underscore, for example, en_US, en_GB.

There are different constants, such as Locale. US, for specifying different locale. By passing these constants as parameters, you can create different locale objects.

ResourceBundle

The `java.util.ResourceBundle` class is used for holding the locale-specific object or properties. You can use this class to retrieve the required locale-specific resources and easily localize programs or translate them into different languages. You simply add the required classes or properties to support more locales.

By passing the name and the locale to the `ResourceBundle.getBundle()` method, you can retrieve the class or property file matching the

`name_language-code_country-code`. If "Test" is the name and the locale is "en_US," the ResourceBundle scans for a class or property file named "Test_en_US." If it is not found, the search continues for the file named "Test_en." If the search fails, the default Test.class or Test.Properties file loads.

Unicode

Unicode defines a standard and universal character set. The JVM uses the Unicode character-encoding standard. Characters and strings are represented by 16-bit character code. Classes such as InputStreamReader and OutputStreamWriter in the java.io package support reading and writing of character data streams using a variety of encoding schemes. The default encoding standard is ISO 8859-1 (ISO Latin_1). If the application uses a character set the default encoding format cannot handle, the encoding standard should be specified explicitly.

Unicode Transformation Format (UTF) is a multi-byte encoding format that stores some characters in one byte and others in two or three bytes. If most of the data is ASCII based, UTF is more compact and is consequently a more widely used character-encoding scheme than Unicode. UTF-8 is an eight-bit form of UTF, the unification of US-ASCII and Unicode. UTF-8 is a variable-width character encoding that encodes 16-bit Unicode characters into one or two bytes. Encoding internationalized content in UTF-8 is recommended by Sun because it is compatible with the majority of existing Web content and provides access to the Unicode character set.

Java.text package

The java.text package has classes and interfaces to handle text, date, number, and messages. These classes use the default locale object. You can overwrite this default locale behavior by specifying your preferred locale. You use these classes for formatting, parsing, searching, and sorting the locale-sensitive information, according to the locale assigned.

InputStreamReader and OutputStreamWriter

The InputStreamReader class reads data from InputStream as bytes and converts them to characters, according to a specified character encoding. The OutputStreamWriter class converts the characters to bytes and sends them to OutputStream. JVM uses the 16-bit Unicode format, and many of the operating systems use 8-bit Unicode format. So, any data entering the JVM must be converted to 16-bit Unicode encoding format, and any data leaving the JVM must be converted to 8-bit Unicode encoding format. While creating Reader or Writer classes, the required encoding can be passed as a parameter. The ISO number represents the encoding. For example, ISO 8859_1 is passed as 8859_1. If nothing is specified, default encoding of the platform is used.

Summary

In this section, we discussed various culturally dependent data, such as text

messages, colors, currencies, and algorithms. These elements are good candidates for internationalization. You also learned that the following APIs help in i18n and l10n.

- **Properties:** Loads values from property files at startup or runtime.
- **Locale:** Identifies the specific locale for a particular session or user.
- **ResourceBundle:** Holds the locale-specific object or properties.
- **Unicode encoding standard:** Defines a standard and universal character set.
- **java.text package:** Handles text, date, number, and messages.
- **InputStreamReader:** Reads data from InputStream as bytes and converts them to characters, according to a specified character encoding.
- **OutputStreamWriter:** Converts the characters to bytes and writes to OutputStream.

Test yourself

Question 1:

You plan to develop a portal in five different languages to support clients from different countries. Which of the following application features need to be configured for internationalizing the application?

Choices:

- **A.** Text of UI elements
- **B.** Authentication routine
- **C.** E-mailing subsystem
- **D.** Currency
- **E.** Color of the pages

Correct choice:

A, **D**, and **E**

Explanation:

Choices A, D, and E are the correct answers.

Text of UI elements, such as the labels of text fields and buttons must be internationalized because they have to be presented in different languages Thus, choice A is correct.

Currency symbol differs from country to country, so choice D is also correct.

Interpretation of the colors depends on the cultural preferences of the country. Some colors might be offensive in a country. Therefore, colors are also a candidate for customization, and choice E is correct.

Although the UI elements required for capturing the authentication information might vary, the actual authentication routine works almost the same, irrespective of the location. Hence, choice B is incorrect.

The bodies of the e-mails may be internationalized, but the basic protocol for mailing remains the same regardless of the location. Therefore, choice C is incorrect.

Section 12. Security

Introduction

Security has become a prime concern these days due to the financial damages caused by malicious hack attacks. As an architect, you should be in a position to minimize these attacks by building a robust security model for your application. In this section, we briefly introduce to the Java 2 security model and the security restrictions on applets. Later, we discuss important security concepts, such as authentication, authorization, symmetric and asymmetric encryption algorithms, and digital signatures and certificates. We end this section with a discussion on topologies.

Java 2 security model

In JDK 1.1, any downloaded code, such as applets, was considered untrusted and consequently run in a restricted sandbox, whereas local applications and signed applets were given full access to the system resources. The Java 2 security model changes this coarse-grained approach to a fine-grained, policy-driven approach. When code is loaded, a policy file is read, and the allowed permissions are granted to the code. The permissions can be anything from read or write access to a directory to connect permission to a host computer. Code can access the resource only if it has been granted the required permissions for accessing that resource. Permission classes are extensible, so custom permissions and properly configured policy files provide the required granularity of security.

Applet security

Applets are Java programs that run inside the Web browser. They are typically embedded in a Web page to add dynamic behavior. Applets are loaded with a

restrictive policy file. The most important restrictions are:

- Making network connections to arbitrary hosts other than the originating host.
- Reading/writing on the client file system.
- Starting other programs on the client.
- Loading native libraries.
- Defining native methods.
- Any operation that could be detrimental to the client system. (This excludes attacks such as excessive usage of CPU, memory, and network resources, as they can be handled at the OS level.)

It is important to know that these restrictions do not apply to applets loaded from the local file system whose classes are present in the client's CLASSPATH.

Security fundamentals

Following are some of the fundamental terminologies you must be familiar with when dealing with security issues:

- **Principal**
  Any identifiable person, role, or a system.
- **Authentication**
  The means by which communicating entities prove to one another that they are acting on behalf of specific identities authorized for access (that is, the process by which a user or a system is identified by the other party). For example, a customer logs in to a bank's Web site using his or her login and password. This combination of user name and password identifies the user to the system.
- **Authorization**
  The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints. For example, a manager can see all the employee details, whereas employees can only see their details.
- **Data integrity**
  The means used to prove that information has not been modified by a third party while in transit. For example, if you send a file and its checksum separately, the receiving party can compute the file's checksum and match it with the received checksum to ensure the file's contents were not tampered with along the way.
- **Confidentiality (data privacy)**
  The means used to ensure that information is made available only to the

users who are authorized to access it. For example, you can encrypt the data and send. The receiver who has the decrypting key alone would be able to read the data.

Cryptography

Cryptography is the practice and study of encryption and decryption -- encoding data so it can only be decoded by intended recipients and rendered unreadable for others. There are two forms of encryption:

- Symmetric
- Asymmetric

Symmetric

Both sender and recipient know a common key used to encrypt and decrypt messages. Because the keys are same for both encrypting and decrypting, it is known as symmetric encryption.

One benefit of this method is:

- Requires significantly less resources in terms of CPU cycles to encrypt and decrypt the data.

A disadvantage of this method is:

- Both the sender and receiver must share the key in a secure way. If it is leaked to a third party, the entire mechanism becomes futile.

Asymmetric

Two different but related keys are used in such a way that one key, called a private key, is kept as a secret, while the other public key is available to anyone. The two fundamental principles that drive this method are:

- One key cannot be deduced from the other.
- Messages encrypted with one key can only be decrypted by the other and vice versa.

One advantage of this method is:

- Completely eliminates the need to securely share the keys, as a sender can use the recipient's public key to encrypt the message, which can be read only by the recipient using his or her secure private key.

One downside of this method is:

- It is computationally expensive.

In reality, you can use both forms of encryption in combination for enhanced security and efficient use. You can use symmetric encryption to encrypt the message, thereby reducing the computational cost involved in decrypting; the shared key (which would be small compared to the data) is encrypted using the asymmetric encryption, eliminating the necessity to transfer the keys securely.

Digital signatures and certificates

You can use asymmetric keys to verify the sender. Let's say if a message can be decrypted using one's public key, it ensures that it was encrypted using the sender's private key. This fact can be used to verify the sender's authenticity because he or she is the only one who has access to his or her private keys (provided it has not been hacked by someone else).

But how do you get an individual's public key, and how do you know that it's really his or her public key? A certificate that contains the name of the individual, expiration date, and a copy of the individual's public key solves this dilemma. A central certificate issuing authority called the Certificate Authority (CA) verifies people's identity and grants them the certificates. The granted certificates are digitally signed by the CA to ensure their credibility. The CA certificates are typically installed, by default, in applications such as Internet browsers. The CA is trustworthy, so the individuals who are trusted by it are also trustworthy. This chain of trust is what makes the entire system function properly.

Certificates are issued in various strengths depending on the level of credibility the CA has on the individual. A certificate of the lowest grade could be obtained by simply proving you have a valid e-mail ID, whereas a commercial grade might require advanced identification techniques such as DNA test.

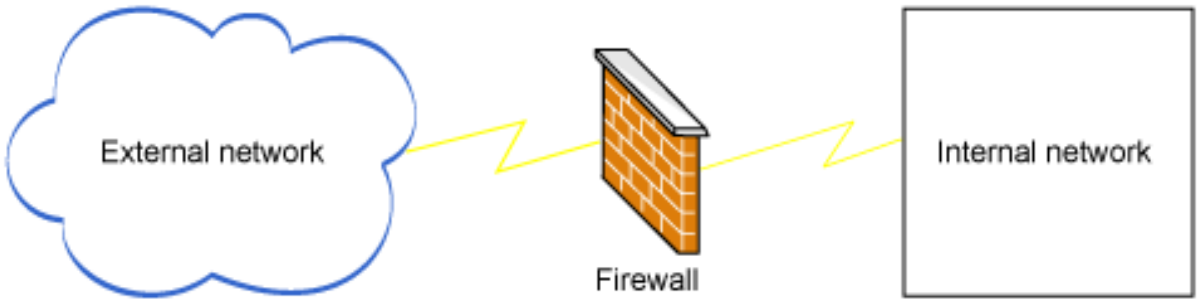
The CA's job does not end with issuing of certificates. It's also in charge of maintaining the certificate's status. If someone's private key is lost, he or she can report the theft to the CA that issued the certificate. CA adds the certificate to the CA Revocation List (CRL) that lists the compromised certificates. Anyone can access this database to ensure that the certificate they trust is not a malicious one.

Network topologies for implementing security

The layout of the network has a strong correlation with the security of the network. Multiple entry points to the network without proper access control mechanisms are a boon for intruders looking to penetrate corporate networks. For enhanced security, the entry points into the network have to be restricted and must be guarded by well-configured firewalls. Once the topology is set, there must be constant monitoring of the firewall, server, and other network equipment log files to uncover any malicious activities, such as unauthorized intrusions, in a timely manner.

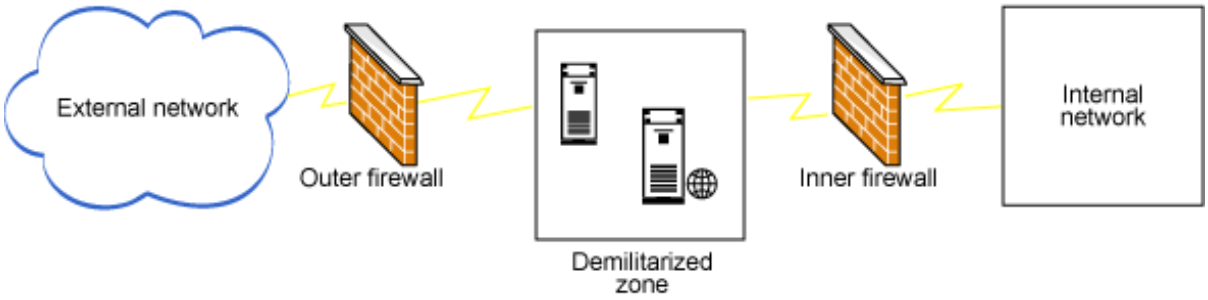
Simple firewall

This is a simple model in which the internal network and the external network are separated by a firewall.



Two firewalls and DMZ

You use this model when you must offer a significant amount of services to the external network. You place the externally accessible servers in a demilitarized zone (DMZ) surrounded by firewalls on either side of the network. You configure the inner firewall more restrictively than the other firewall. Any communication from the external network to the internal network happens only through the servers deployed in the DMZ.



Tunneling

The firewall setup does not generally allow every protocol to communicate through it. Opening up numerous ports can result in an extremely vulnerable firewall. So, administrators generally allow only well-defined protocols, such as HTTP and HTTPS.

You can use tunneling to access an external service that is not allowed by the firewall by piggy-backing the requests onto a protocol that is allowed by the firewall (for example, using HTTP as a covert channel for invoking Web services).

Similarly, external networks can tunnel into an internal network. But this is not good practice as it allows anyone with malicious intentions to bypass the firewall rules.

Summary

In this section, we discussed the security model of the Java language and the fundamentals of security required to face this exam. Remember the security restrictions imposed for an applet. Also, try to understand the various basic concepts of security, such as cryptography, signatures, firewalls, DMZ, and tunneling. The

exam does not require you to know, in detail, the algorithms and mechanisms; rather, it tests your overall knowledge on the concepts and terminologies.

Test yourself

Question 1:

Which of the following statements are true about a DMZ?

Choices:

- **A.** A DMZ is the zone secured behind a firewall.
- **B.** A DMZ is the zone before a firewall.
- **C.** A DMZ is the zone in front of two firewalls.
- **D.** A DMZ is the zone between 2 firewalls.

Correct choice:

D

Explanation:

Choice D is the correct answer.

A DMZ is the zone between two firewalls. Hence, choice D is correct. The remaining choices are incorrect as they do not describe a DMZ.

Section 13. Wrap-up

Points to remember

We conclude this tutorial with some crucial points to remember. The SCEA exam is completely different from other exams in terms of its breadth. An architect is required to know about numerous topics and must decide on the correct solution for a given problem. So, you must learn many concepts, and there isn't any single book that can cover all the subjects required for the exam.

Due to the vastness of the topics covered, the exam might appear difficult. But if you are an experienced architect or willing to spend a significant amount of time understanding the basics, this will be a fairly easy exam. There is nothing to memorize and everything is at a conceptual level. So, don't expect to see any programming questions.

Because this is one part in a series of exams, you won't receive a certificate when you complete Part 1. You must complete the assignment (Part 2) and the essay (Part 3) to finish the SCEA track.

We hope this tutorial was helpful in your exam preparation and wish you all the best for your exam.

Resources

Learn

- SCEA certification study guides help you focus on the exam objectives:
  - *Sun Certified Enterprise Architect for J2EE Technology* by Mark Cade and Simon Roberts (Prentice Hall PTR, 2002)
  - *Sun Certified Enterprise Architect for J2EE Study Guide (Exam 310-051)* by Paul R. Allen and Joseph J. Bambara (McGraw-Hill Osborne Media, 2003)
- The following books give you a greater understanding of the exam objectives:
  - *UML Distilled* , by Martin Fowler and Kendall Scott (Addison-Wesley, 1999)
  - *Design Patterns - Elements of Reusable Object-Oriented Software* , by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995)
  - *Java design patterns 101* by David Gallardo (developerWorks, January 2002)
  - *Java design patterns 201* by Paul Monday (developerWorks, April 2002)
  - *Mastering Enterprise JavaBeans* by Ed Roman (Wiley, 2001)
  - *Getting started with EJB technology* by Joe Sam Shirah (developerWorks, April 2003)
  - *Java Message Service* by Richard Monson-Haefel and David Chappell (O'Reilly, 2000)
  - *Introducing to the Java Message Service* by Willy Farrell (developerWorks, June 2004)
  - *Java Internationalization* by David Czarnecki Andy Deitsch (O'Reilly, 2001)
  - *Java internationalization basics* by Joe Sam Shirah (developerWorks, April 2002)
  - *Java Security* by Scott Oaks (O'Reilly, 2001)
  - Java security Part 1: Crypto basics and Part 2: Authentication and authorization by Brad Rubin (developerWorks, July 2002)
- Prepare for all the dimensions of Java certification with the other tutorials in this series:
  - *Java certification success, Part 3: SCBCD* by Seema Manivannan and Pradeep Chopra (developerWorks, September 2004)
  - *Java certification success, Part 2: SCWCD* by Seema Manivannan

(developerWorks, May 2004)

- *Java certification success, Part 1: SCJP* by Pradeep Chopra (developerWorks, November 2003)

- Whizlabs founder Pradeep Chopra offers additional guidance with SCEA certification in his article "Programming Success: The SCEA Certification."
- Don't miss the SCEA FAQs on JavaRanch.
- Another useful feature on JavaRanch is Architect Certification Forum.
- You'll find articles about every aspect of Java programming, including all the concepts covered in this tutorial, in the developerWorks Java technology zone.

Discuss

- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

Sivasundaram Umapathy
Sivasundaram Umapathy holds a bachelor of engineering degree in computer science from the University of Madras and a master of science degree in software systems from BITS, Pilani. He is presently associated with Sella Synergy India Limited, India, the software division of Banca Sella, S.p.A, where he designs and develops mission-critical banking applications using the BEA WebLogic application server. He is also crazy about certifications, with SCJP, SCBCD, SCWCD 1.4, SCMAD, SCEA, OCA, BEA WL7, IBM, and PMP certifications to his credit. He has authored the Whizlabs SCWCD 1.4 and co-authored the Whizlabs SCMAD exam simulators. He actively participates in the open source movement in his free time and is in an expert group member of JSR 244 (J2EE 5.0) and JSR 245 (JSP 2.1).
Acknowledgements: I wish to thank my friend and colleague Mrs. Rajeswari for her support and suggestions in completing this tutorial.