

# The SOA Magazine

## Feature Article



### **Service-Orientation and Object-Orientation Part II: A Comparison of Design Principles**

by Thomas Erl

Published: March 13, 2008 (SOA Magazine Issue XVI: March 2008, Copyright © 2008)

[Download this article as a PDF document.](#)

#### *Abstract:*

*This two-part article series studies object-orientation and service-orientation by providing a comparison of goals, concepts, and principles. Both articles are comprised of excerpts from the book "SOA: Principles of Service Design" [REF-1]. If you haven't already, be sure to read the first part of this article series [REF-5] before continuing.*

*Note: The terms "object-oriented analysis and design" (OOAD) and "object-orientation" are used interchangeably in this article. The upcoming sections contain references to service-orientation design principles that are not explained. If you are new to service-orientation, visit SOAPrinciples.com [REF-4] for introductory descriptions of each of the eight design principles.*

#### **Introduction**

The object-orientation design paradigm is comprised of a rich set of fundamental and supplemental design principles that structure and organize object logic within and across classes. Several of these principles have been carried over into service-orientation to varying extents, while others have been omitted entirely.

This article discusses how service-orientation relates to each of the following object-oriented design principles:

- Encapsulation
- Inheritance
- Generalization and Specialization
- Abstraction
- Polymorphism
- Open-Closed Principle (OCP)
- Don't Repeat Yourself (DRY)
- Single Responsibility Principle (SRP)
- Delegation
- Association
- Composition
- Aggregation

By understanding the relationship between object-oriented and service-oriented design we can identify several specific origins of individual service-orientation principles. More importantly, though, we can establish how specifically services are designed differently from objects.

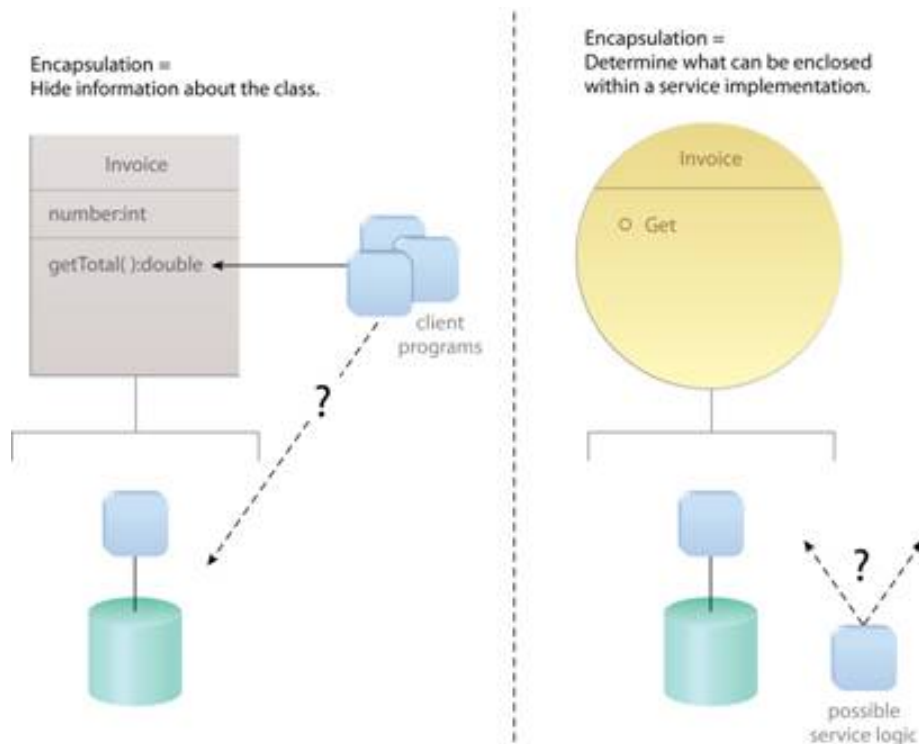
## Encapsulation

To encapsulate means to enclose something in a container. Within object-orientation, *encapsulation* is associated with information hiding and the congregation of supporting informational constructs into a logical whole. It is a principle that states that an object should only be accessed via a public interface and that its implementation should remain hidden from other objects.

The object is the container.

The object-orientation encapsulation principle is comparable to the service-orientation Service Abstraction principle [REF-1], which is also concerned with the deliberate hiding of information.

Services still encapsulate logic and implementations, just as objects do (because, like objects, services are containers). However, within service-orientation, the term "encapsulation" is used more so to refer to what is being enclosed within (encapsulated by) the container. In fact, service encapsulation is related to a fundamental SOA design pattern that is applied to determine what logic is and is not suitable as part of a given service.



*Figure 1: Although both revolve around the same meaning behind encapsulation, each design paradigm uses the term somewhat differently.*

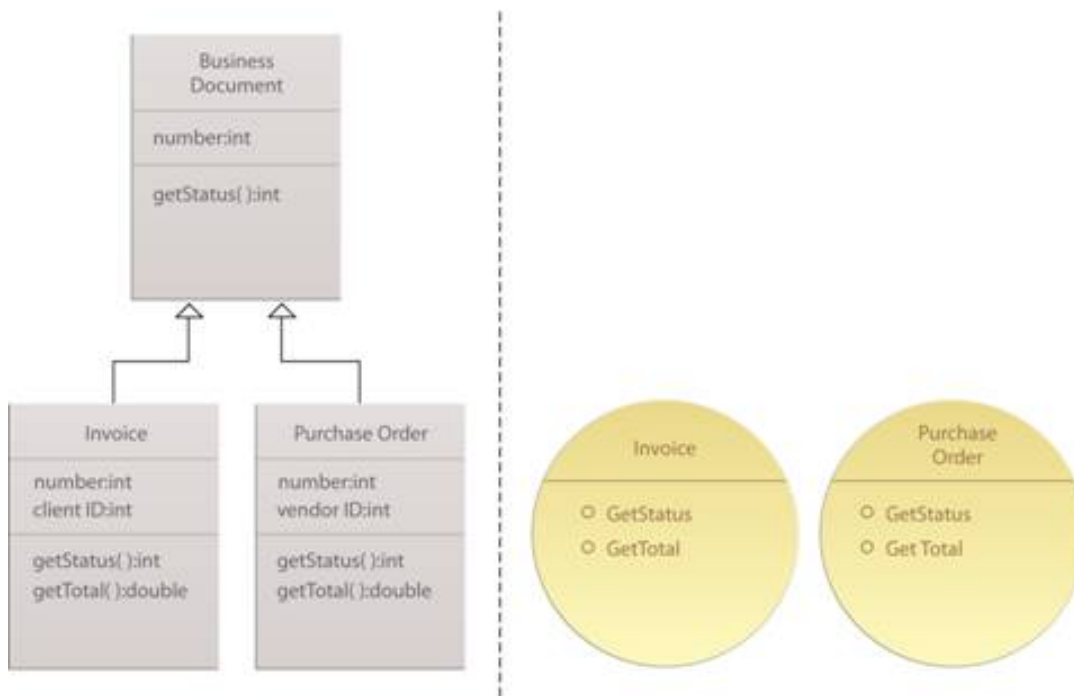
## Inheritance

A primary means by which object-orientation aims to achieve code reuse is by organizing logic into classes and then establishing relationships between these classes. Various types of relationships can exist, the most formal of which is *inheritance*.

Two classes can form a parent-child relationship, where the child class is automatically assigned (inherits) the methods and attributes of the parent class. When two classes are associated in this manner, the parent class is referred to as the super-class of the child class, and the child class is the sub-class of the parent class. A sub-class can do anything the super-class can do, plus it can be further extended with unique functionality (through a process called specialization).

The concrete bond formed between a super-class and its sub-class is often labeled as an "is-a" relationship because whatever the sub-class exists as is an implementation of what is defined in the super-class. This relationship is

expressed using a white, triangular arrow head, as shown in Figure 2. Due to the emphasis on individual service autonomy and reduced inter-service coupling, inheritance between services is generally discouraged within service-orientation. And, because services do not formally implement each other, they are not required to establish "is-a" relationships.



*Figure 2: With inheritance, sub-classes (bottom left) representing specific types of business documents can implement an abstract super-class (top left) to inherit a common method and attribute. Entity services (right) representing business documents may share similar capabilities, but none are inherited.*

**Note:** In some circles it has become an OOAD convention to not repeat inherited attribute and method names in sub-classes because their existence can be assumed via the expressed inheritance relationship. Inherited attributes and methods are intentionally displayed in all examples in this chapter to more clearly establish a comparison with corresponding service definitions.

It is also worth noting that up interface inheritance is possible within Web services, as of version 2.0 of the WSDL specification via the interface element's extends attribute [REF-2].

## Generalization and Specialization

A well designed top-level super-class (also referred to as the abstract or base class) expresses a highly generic interface with broad applicability. This allows for the definition of a range of sub-classes.

Generalization is achieved when a parent super-class is defined. Because sub-classes implement distinct (specialized) variations of a super-class, their definition is referred to as *specialization*. Generalization is another way of saying a class has a "is-a-kind-of" relationship with another class, whereas specialization represents the previously described "is-a" relationship.

There are concepts similar to generalization and specialization within service-orientation, only because inheritance is not supported, they exist differently. Within the context of service design, generalization and specialization relate directly to granularity. The more specialized a service, the greater its degree of service-level granularity.

Determining the right degree of specialization for each service is a critical decision point and one that establishes a service's functional context and concrete boundary. However, through the use of service design patterns [REF-3], an existing coarse grained (more generalized) service can be decomposed into finer-grained (more specialized) services for functional and practical reasons, but not as a result of inheritance.

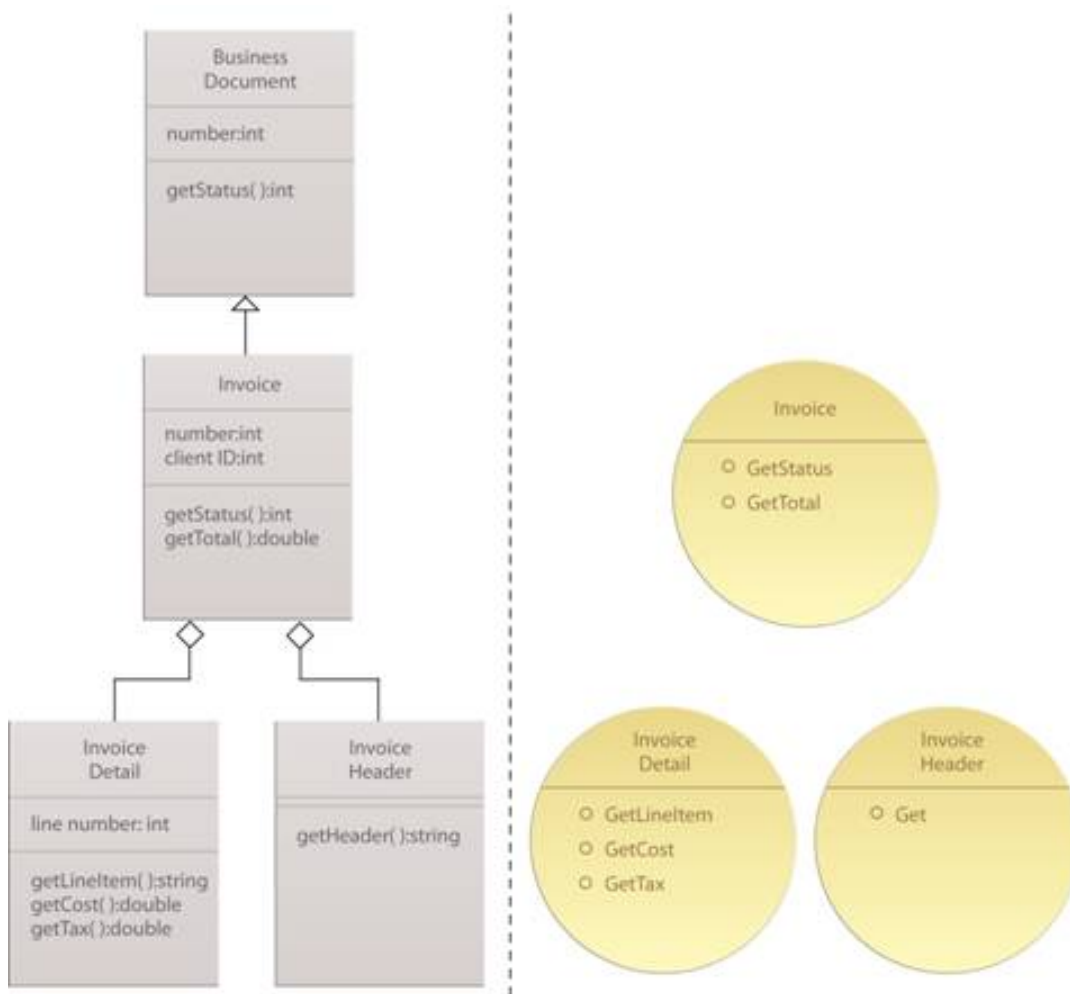


Figure 3: The three-level hierarchy (left) includes the Invoice class, which is a specialization of the generalized Business Document class, and the Invoice Detail and Invoice Header classes, which are aggregates of the Invoice class. Technically, the Invoice Detail and Invoice Header classes are not considered specialized because they are not based on "is-a" relationships. Alternatively, because the Invoice Detail and Invoice Header services (right) have a higher level of service granularity than the Invoice service, they can be considered more specialized (but not in a traditional OOAD sense). (Note the use of the diamond symbol to indicate an aggregated relationship. This is explained further in the upcoming Aggregation section.)

**Note:** The aforementioned base or abstract classes defined through the creation of generalized classes are not actually implemented in the real world, in that instances or objects are not generated from these classes. Instead, they exist to establish inheritance structures and specialized sub-classes. In service-orientation the use of an abstract class is voluntary, as explained at the end of this article.

## Abstraction

Another information hiding related principle in object-orientation is that of *abstraction*. Specifically, the purpose of abstraction is to create a simplified class that hides the complexity of the underlying implementation and exposes only the most necessary (abstract) methods and attributes. Abstraction can be applied to support inheritance for the definition of abstract classes that are not implemented but instead form the parent super-class from which numerous specialized sub-classes can be defined and implemented.

Conceptually, object-oriented abstraction is similar to Service Abstraction [REF-1] in that both principles ultimately intend to streamline public information about underlying solution logic and implementation details. However, because service-orientation does not support inheritance, there is no corresponding notion of an abstract class.

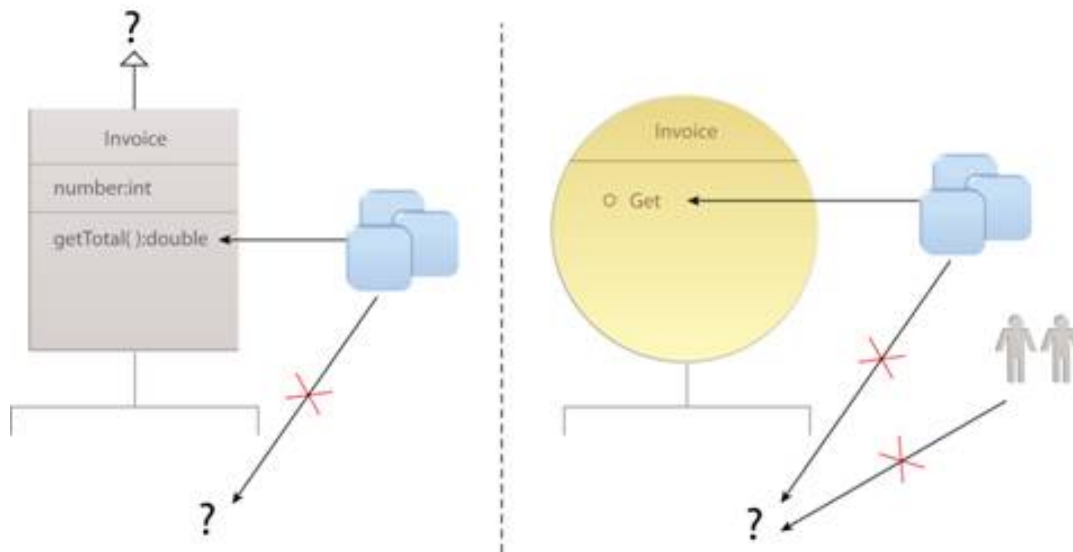


Figure 4: While object-oriented abstraction (left) is primarily concerned with hiding complexity (in this case, the underlying implementation) from other consumer programs, Service Abstraction (right) also limits human access and awareness of underlying service details.

## Polymorphism

When multiple object-oriented sub-classes inherit and retain a method from a super-class, you can end up with multiple classes that have identically named methods. Even though the method definitions are the same, the implementation will vary across the sub-classes because each sub-class is specialized in a distinct manner. Therefore, the same message sent to any one of these sub-classes will have different results based on the variance of the sub-class implementations. This is known as *polymorphism*.

Because inheritance does not exist in service-orientation, this form of polymorphism is also not applied to individual services. The closest thing to polymorphism that can be realized in support of service-orientation is the consistent functional expression of service contracts, as per the Standardized Service Contract principle [REF-1]. This typically results in similar or identically named capabilities across numerous services. (The consistent use of standardized, verb-based CRUD-style operations within entity services is an example of this.)

This level of interface consistency is the result of naming conventions applied to the contract designs. There is no expectation that identically named capabilities of different services support the same messages. Therefore, this would not qualify as true polymorphism.

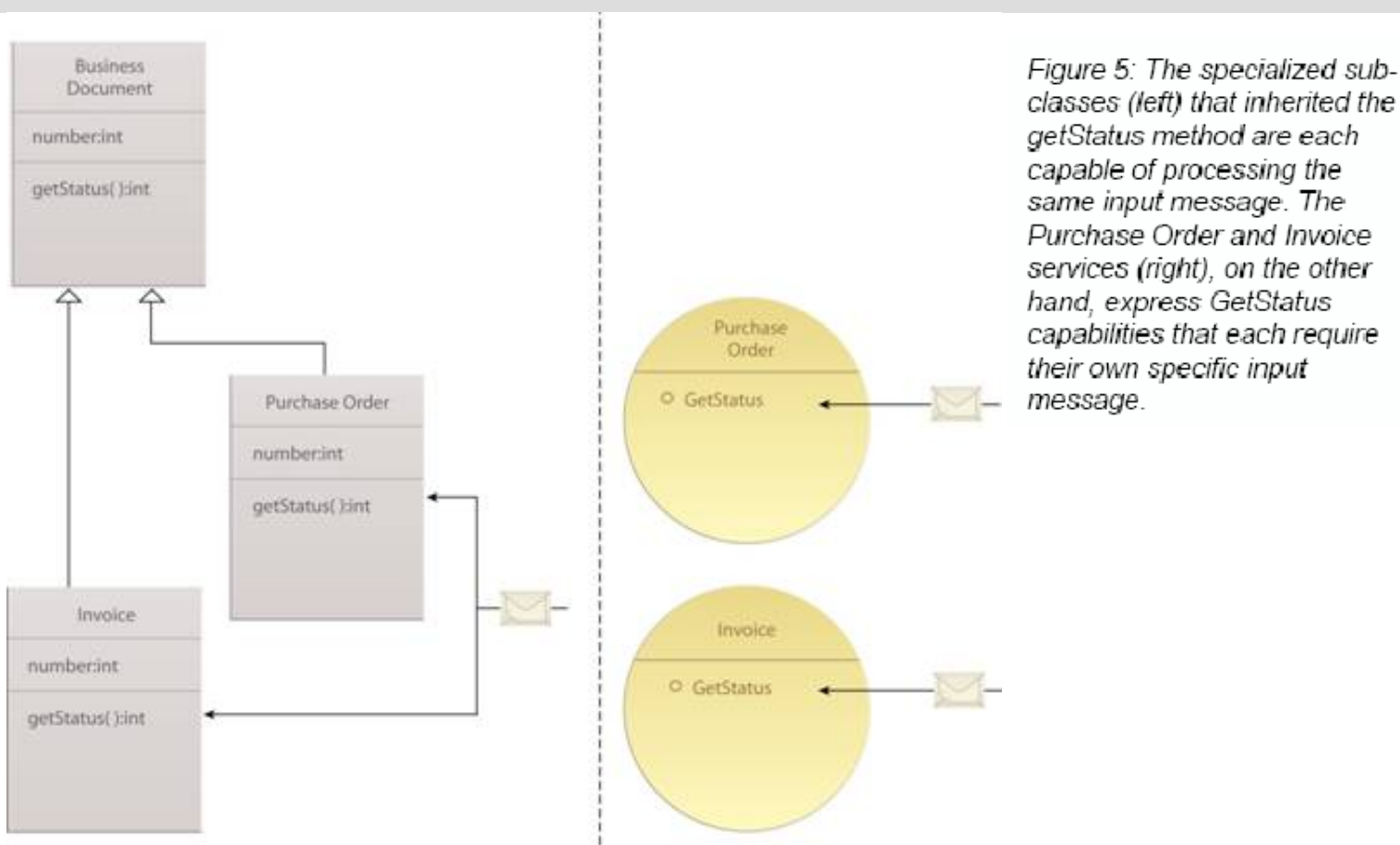


Figure 5: The specialized sub-classes (left) that inherited the `getStatus` method are each capable of processing the same input message. The `Purchase Order` and `Invoice` services (right), on the other hand, express `GetStatus` capabilities that each require their own specific input message.

### Open-Closed Principle (OCP)

This basic design principle states that classes should allow (be open to) extension, but disallow (be closed to) modification of what has already been implemented. This is a key design requirement that helps protect reusable functionality upon which multiple client programs have already formed dependencies. It is fully applicable to service contracts and is required when applying the Service Reusability principle [REF-1] in order to minimize subsequent governance burden.

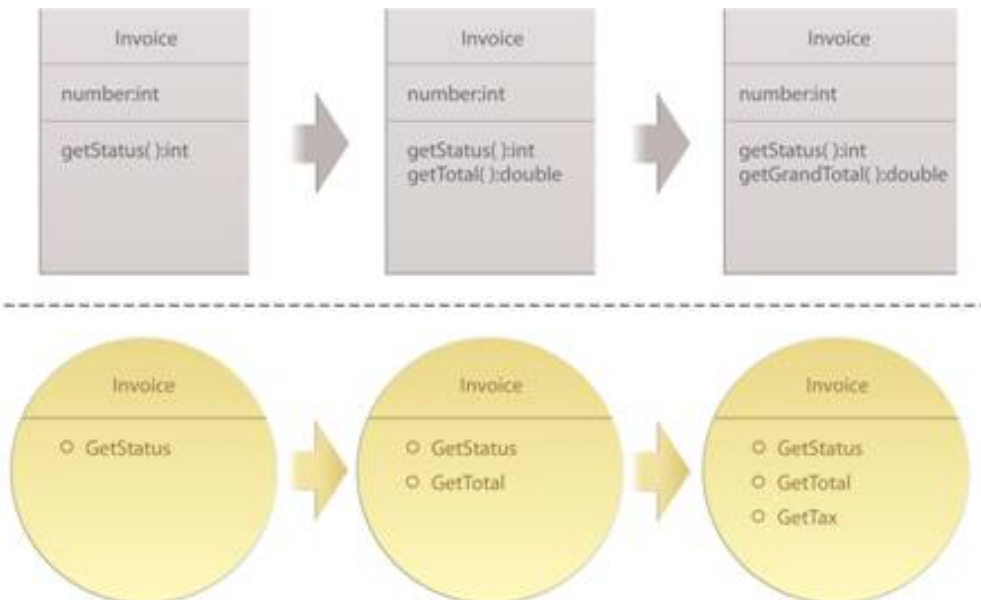


Figure 6: The class on the top right violates this principle by renaming (not overriding) an already implemented method, whereas each of the services on the bottom comply with this principle by only extending the service contract.

## Don't Repeat Yourself (DRY)

By avoiding redundant code, objects can be more effectively reused and wasted development effort can be minimized. This principle simply states that if reusable logic exists, it should be separated so that it can be made available for reuse. The rationale behind this principle forms the basis of the Service Normalization pattern [REF-3] and the use of agnostic service models. By avoiding functional overlap we furthermore avoid redundancy across service designs in support of the Logic Centralization pattern [REF-3].

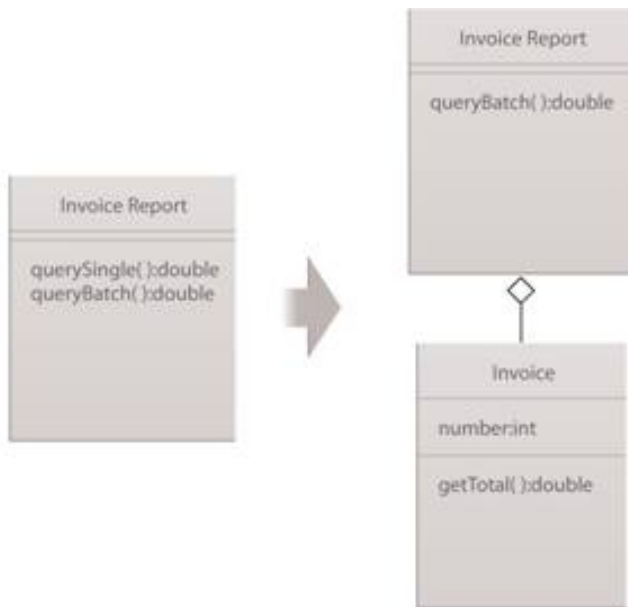
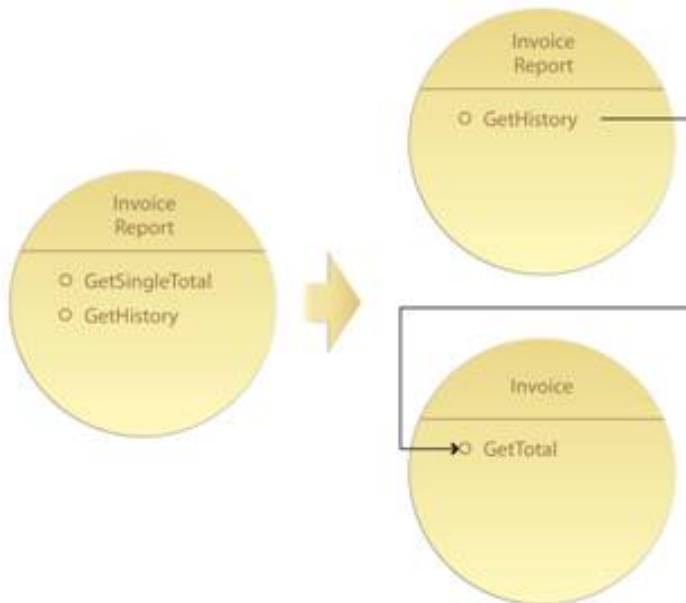


Figure 7: The class and the service on the left are each further decomposed so as to extract reusable logic into a separate agnostic functional context represented by the class and service on the bottom right.



## Single Responsibility Principle (SRP)

Object-oriented units of solution logic designs are ideally centered around a single overall purpose. The single responsibility principle encourages us to limit their functional scope to this purpose, so that they are only required to change if that one purpose changes.

In service-orientation, this corresponds to the consistent use of service models that establish distinct functional service contexts. An entity service, for example, can be dedicated to processing associated with a business entity. Similarly, a task service's sole purpose is the automation of a specific business process.

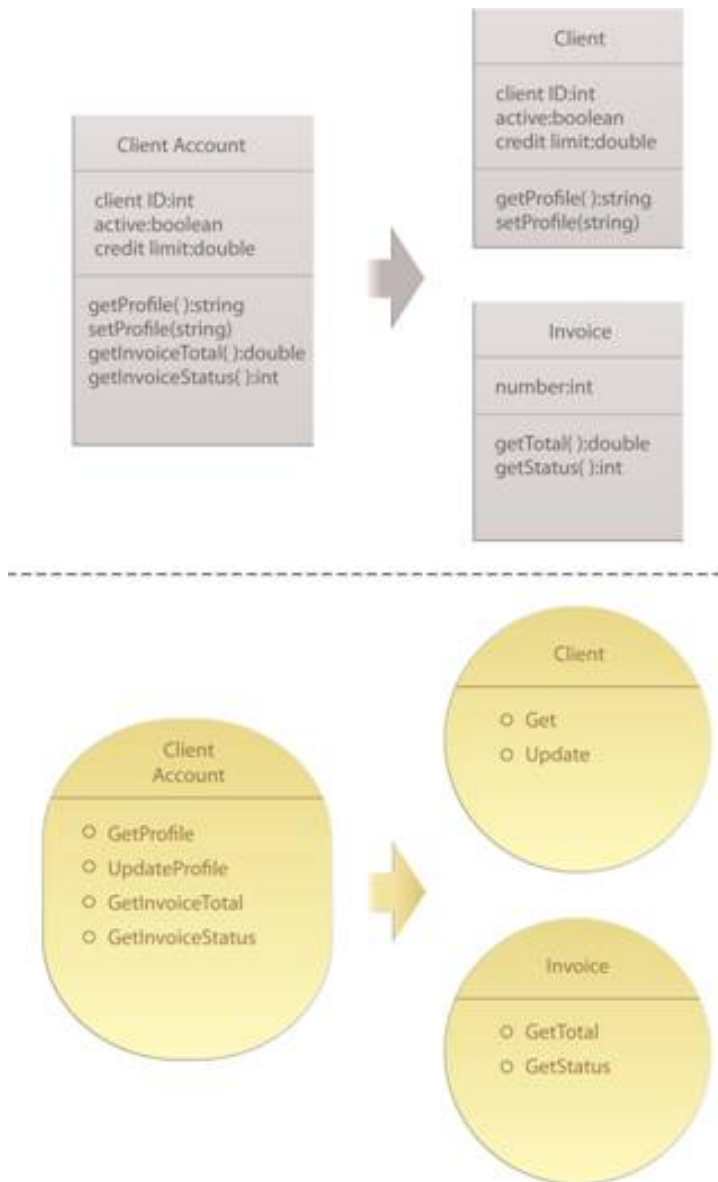


Figure 8: The coarse grained, dual-purpose Client Account class and service (left) are each decomposed into two groups of behaviors that represent two distinct, single purpose functional contexts (right).

The single responsibility principle is closely related to the notion of cohesion. A class or service has increased cohesion when it defines (and sticks to) a specific functional context. This establishes it as a container for a group of highly related methods or capabilities. Alternatively, a class or service with low cohesion is one that violates this principle by defining a functional context that encompasses multiple purposes.

Note that cohesion and service granularity don't always go hand-in-hand. A service can have a coarse level of service granularity, and still be highly cohesive. However, because the functional scope of services that facilitate numerous purposes or responsibilities naturally tends to result in lower (broader) granularity, services with low cohesion are usually more coarsely grained.

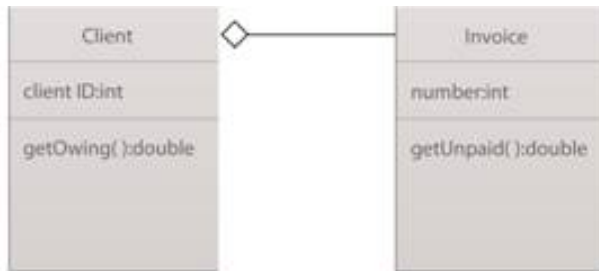
**Note:** In past chapters we've made reference to positioning services as multi-purpose enterprise resources in support of the Service Reusability principle [REF-1]. In this context, the term "multi-purpose" refers to the utilization of a single service (or any one of its capabilities) within multiple usage scenarios. The service itself retains a specific functional context and boundary and therefore can be said to have a single responsibility.

## Delegation

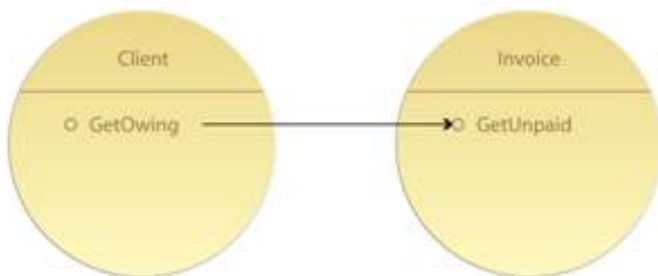
This simple principle states that if an object requires logic that already exists in another object, then it should give (*delegate*) the responsibility of carrying out that logic to the other object instead of carrying it out itself. The key condition of applying this principle is that the behavior of the object to which the responsibility is being delegated not

be required to change.

Consistent use of this fundamental design consideration fully supports Service Reusability and the realization of widespread Logic Centralization [REF-3]. In fact, delegation directly corresponds to the fundamental service design patterns that require the invocation and reuse of logic external to a service's functional context so as to preserve the integrity of this context. In service-orientation, the rationale behind this principle is what drives the need for service composition.



*Figure 9: Instead of carrying out the retrieval of a range of invoice data on its own, the Client class delegates the responsibility by invoking a corresponding Invoice class. The Client service's GetOwing capability does the same by invoking the Invoice service's GetUnpaid capability.*



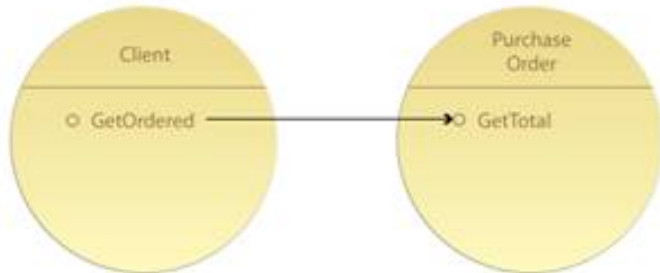
## Association

In OOAD, an *association* between two classes represents a relationship. Relationships are required to carry out delegation so that objects can invoke and communicate with each other at runtime. There are different types of associations, the simplest of which establishes a "uses-a" relationship that allows one class to exchange messages with another. The two classes can be unrelated and independent; one just uses functionality the other has to offer.

Other, more formal types of associations define different types of relationships. Aggregation and composition, for example, create "has-a" relationships between classes that have ownership implications (as explained in the upcoming sections). Service interaction is very similar to the former type of association in that services only need to be able to use each other's capabilities without ownership-related limitations. This is why service interaction is typically identified using the same (or similar) arrowhead as used to express object-oriented association relationships.



Figure 10: The Client class has an association with the Purchase Order class (top) similar to how the Client service relates to the Purchase Order service (bottom).



## Composition

The concept of *composition* in object-orientation and service-orientation is similar but, as with encapsulation, the term is used differently. In OOAD composition refers to a form of association that establishes an ownership structure between classes. A parent class is composed of others and therefore creates "has-a" relationships with other classes.

Furthermore, composed objects have a lifespan associated with the parent object, meaning that they are destroyed when the parent object ceases to exist. Composition relationships are identified with lines that end with a black diamond shape attached to the class responsible for initiating the composition.

In service-orientation, the term "composition" refers to an assembly or aggregate of services with no predefined ownership structure. Therefore, the rules associated with OOAD composition do not apply to service-orientation. Services are free to invoke capabilities within other services and the composition controller service instance responsible for initiating the composition does not need to remain active for as long as any of the composition member service instances. This level of freedom is important to fully realize the potential of Service Composability.

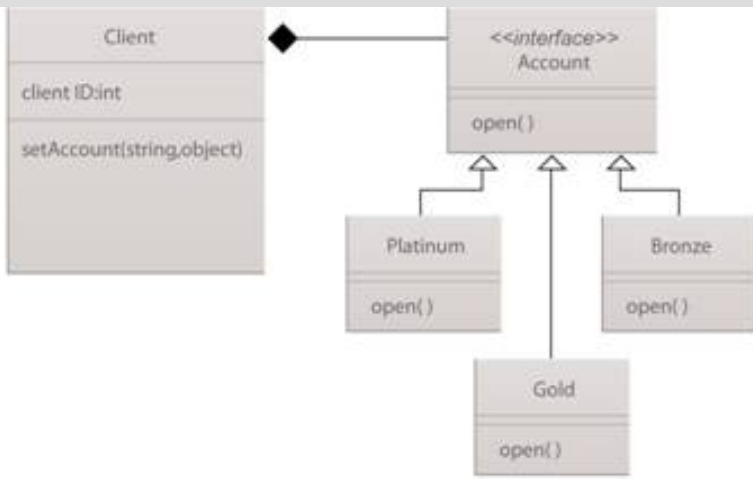
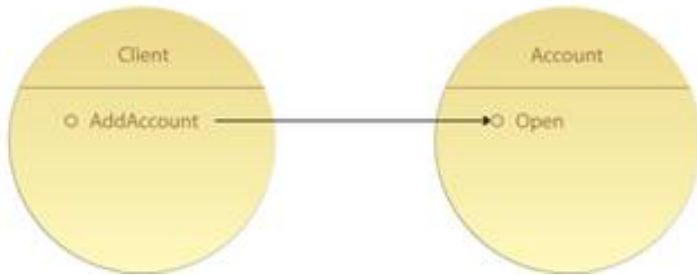


Figure 11: The Client class composes a related Account class which is implemented by one of three sub-classes depending on the nature of the account (top). In this case, the Account interface is owned by the Client class and cannot exist without it. The Account service, on the other hand, is simply invoked by the Client service with no ownership implications (bottom).



### Aggregation

The object-orientation principle of *aggregation* is similar to that of composition, but different rules apply to the relationships between participating objects. Classes with an aggregation association still establish an ownership structure based on a "has-a" relationship. However, the lifespan of the object that initiates an aggregation does not need to equal the lifespan of the other participating objects. In other words, the class being aggregated is allowed to exist independently outside of the parent (container) class acting as the aggregator. Aggregation relationships are distinguished by a line with a white diamond touching the class that is aggregating others.

As with composition, aggregation does not apply to service-orientation because it is still based on a "has-a" ownership structure. As mentioned in the Association section, service interaction most closely resembles a "uses-a" relationship.

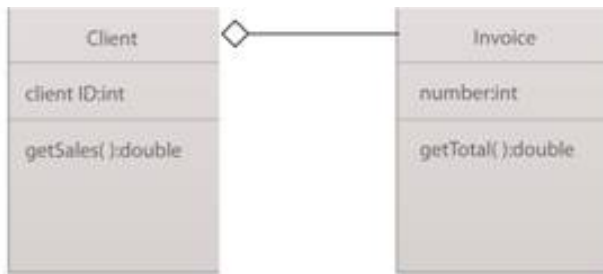
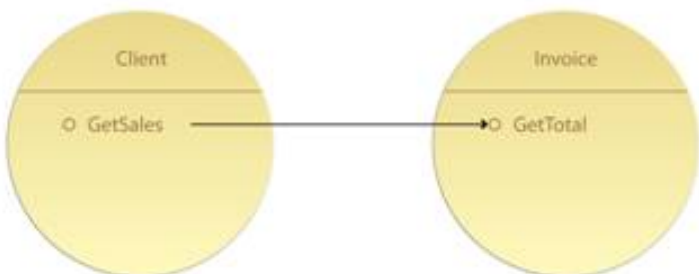


Figure 12: The Client class aggregates the Invoice class (top) because Clients have invoices and invoices can exist independently from clients. However, the Client service invokes the Invoice service (bottom) as it would any other service.



## Guidelines for Designing Service-Oriented Classes

To conclude this study of service-orientation and object-orientation principles, it is worth framing service-orientation design considerations within the context of OOAD conventions and class design.

The remaining sections provide a set of guidelines for designing service-oriented classes. These guidelines can be helpful should you be required to model services using the UML class notation.

### *Implement Class Interfaces*

Classes positioned as services should always implement interfaces so that an official public contract is expressed separately from additional class details that may need to remain hidden. This directly supports the Standardized Service Contract, Service Loose Coupling, and Service Abstraction principles [REF-1].

### *Limit Class Access to Interfaces*

This guideline is essentially a translation of the Contract Centralization pattern and the consumer-to-contract coupling type defined by the Service Loose Coupling principle [REF-1]. This positive form of coupling protects the underlying class implementation details the same way it prevents negative forms of coupling within services that exist as Web services.

### *Do Not Define Public Attributes in Interfaces*

This already exists as a best practice in OOAD, but is worth repeating here in support of service-orientation. The Service Statelessness principle [REF-1] encourages services to exist as solution units capable of reverting to a stateless condition whenever appropriate. Removing attributes from the public interface forces all communication through methods (be they accessor methods or otherwise) and therefore places the control of how state is managed within the service (which is exactly where we want it).

### *Use Inheritance with Care*

Inter-service inheritance is not formally advocated by service-orientation in support of realizing the independence and freedom we seek to establish in every service via the Service Loose Coupling, Service Autonomy, and Service Composability principles [REF-1]. Intra-service inheritance (the application of inheritance to classes encapsulated by the service) can be applied to strengthen the structure of intra-service logic, as required. However, there needs to be a constant awareness that coarse grained services may need to be decomposed into finer grained (more specialized) services at some point. As per the corresponding service design patterns, we can prepare for service decomposition by how we design a service's contract and logic. Service logic comprised of components tightly bound through inheritance structures will be more difficult to decompose into physically separate services than if the underlying class structures are less inter-dependent.

### *Avoid Cross-Service "has-a" Relationships*

Service compositions require the freedom to allow composition members to act independently from the parent controller, even if that means they remain active after the controller instance is destroyed. Furthermore, services can't be limited to some form of pre-determined ownership hierarchy; as per the Service Composability principle [REF-1], a service ideally needs to be able to compose or be composed by any other service within a given inventory.

Unless there is a need for strict rules around the association of object lifespans to parent objects, "uses-a" associations are a more "service-friendly" means of composing classes than composition or aggregation.

### *Use Abstract Classes for Modeling, Not Design*

As explained in the *Generalization and Specialization* section, the use of abstract classes within service-orientation is not required. Because no formal inheritance relationships are defined, no base or abstract service is required for other services to be designed.

However, the use of abstract classes can be helpful during the service-oriented analysis phase (especially for those familiar with OOAD). Abstract classes can be informally defined as the base or root of collections of related classes so as to ensure consistency in the definition of service candidate functional contexts and service capability candidates. As such, their use can be incorporated into a customized variation of the service modeling process.

### *Use Façade Classes*

We haven't discussed the use of façades in this article because they technically represent an OO design pattern (as opposed to a design principle) that corresponds to the Service Facade pattern [REF-3]. However, it is worth mentioning here that from a service design perspective, creating façade classes is a very important and common technique for structuring components as standalone services or as part of service-oriented Web services.

### **References**

[REF-1] "SOA: Principles of Service Design", Thomas Erl, [www.soabooks.com](http://www.soabooks.com)

[REF-2] WSDL 2.0 Primer, W3C, [www.w3.org/2002/ws/desc/wsdl20-primer](http://www.w3.org/2002/ws/desc/wsdl20-primer)

[REF-3] "SOA Design Patterns", Thomas Erl, Prentice Hall/PearsonPTR, pre-release manuscript available at [www.soapatterns.org](http://www.soapatterns.org)

[REF-4] SOAPrinciples.com, [www.soaprinciples.com](http://www.soaprinciples.com)

[REF-5] "Service-Oriented and Object-Oriented Part I: A Comparison of Goals and Concepts", Thomas Erl, The SOA Magazine Issue XV, [www.soamag.com/115/0208-4.asp](http://www.soamag.com/115/0208-4.asp)

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Glossary](#) [SOA School](#) [SOA Books](#) [About](#) [Legal](#)

Copyright © 2006-2008  
SOA Systems Inc.