Chapter 7

Service Coupling (Intra-Service and Consumer Dependencies)

7.1 Coupling Explained

7.2 Profiling this Principle

7.3 Service Contract Coupling Types

7.4 Service Consumer Coupling Types

7.5 Service Loose Coupling and Service Design

7.6 Risks Associated with Service Loose Coupling

7.7 Case Study Example

When assembling the pieces of a machine with nuts and bolts, you want to tighten each part just the right amount. If you over-tighten one, you risk stripping the bolt. If you don't tighten it enough, the machine won't be robust. Each tightened bolt represents a coupling between two parts.

Along those same lines, we need to pay attention not just to where service coupling occurs, but the extent to which the parts of a service composition as well as the parts that comprise its individual services should be coupled.

7.1 Coupling Explained

The term "coupling" is a pretty straight-forward part of the IT vocabulary: Anything that connects has coupling and coupled things can form dependencies on each other. However, when we qualify coupling with "loose" or "tight," we get into a more ambiguous realm. Who's to say what is considered more or less dependent? We will need a firm understanding of how to measure and allocate appropriate levels of coupling (Figure 7.1) in order to effectively apply this principle. Let's therfore begin with a brief exploration of how coupling relates to automation environments in general.

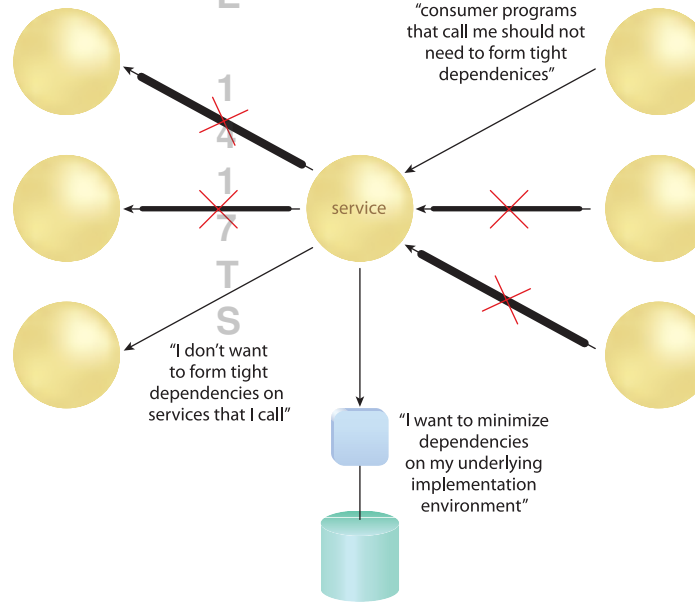

Figure 7.1
This principle emphasizes the reduction ("loosening") of coupling between the parts of a service-oriented solution, especially when compared to how applications have traditionally been designed. Specifically, loose coupling is advocated between a service contract and its consumers and between a service contract and its underlying implementation.

NOTE

In this chapter thicker arrow lines represent tighter coupling requirements.

Coupling in Abstract

Any part of an automation environment that's separable has the potential (and usually the need) to be coupled to something else for the sake of imparting its value. The root of the term (couple) itself implies that two of something exist and have a relationship.

The most common way of explaining coupling is to compare it to dependency. A measure of coupling between two things is equivalent to the level of dependency that exists between them. For example, the relationship between one software program and another represents a measure of coupling associated with interoperability. Or the relationship a technical contract has to the solution logic it is representing indicates a measure of coupling associated with the behind-the-scenes structure of the software program.

This also highlights the fact that the directionality of coupling can vary as well. Two applications tightly coupled by a point-to-point integration channel may have formed a bidirectional dependency in that each application requires the existence (and perhaps even the availability) of the other to function properly. Alternatively, unidirectional coupling is also common where one program may depend on another, but the reverse is not true. The relationship between applications and databases, for example, is a common form of unidirectional coupling (an application may depend on a database, but the database may not depend at all on the application).

Coupling is unavoidable. What we are most interested in when exploring coupling within IT automation is how close this relationship actually is or should be.

Origins of Software Coupling

In the past, many custom applications were developed with certain types and levels of coupling that were simply pre-set by the programming environments or surrounding technology architectures. More often than not, coupling between software programs or components was tight.

For example:

- In a traditional two-tier client-server architecture, the clients were developed specifically to interact with a designated database (or specific process servers).

Proprietary commands were embedded within the client programs, and changes to this binding affected all client installations.

- In a typical multi-tier component-based architecture, components were often developed to work with other specific components. Even shared components that became more popular after OO principles were applied still required tight levels of coupling when made part of inheritance structures.
- When Web services emerged, they were often mistakenly perceived to automatically establish a looser form of coupling within distributed architectures. While Web services can naturally decouple clients from proprietary technology, they can just as easily couple client programs to many other service implementation details.

Interestingly, it is one of the earliest architectures that implemented a more loosely coupled paradigm. Mainframe environments imposed little dependencies on client terminals, allowing the same terminal to be used for multiple types of mainframe applications (a loosely coupled client-server relationship that was later revived with the browser and Web server).

However, while there is some conceptual commonality in this comparison, it's important to point out that loose coupling is a very specific design characteristic that service-orientation aims to establish across all services, especially throughout solution back-ends. Even though mainframe environments had loosely coupled workstation clients, their server-side applications were monolithic and self-contained. Because they weren't typically distributed, there was little emphasis on regulating back-end coupling.

Loose coupling, as a design concept, has historically been a greater part of commercial software design. The use of drivers, for example, allowed client programs to tightly bind to the driver software, which, in turn, decoupled the client programs from underlying hardware and system programs. Similarly, database connection protocols and associated software (such as ODBC and JDBC) also introduce a loosely coupled relationship between the client programs and underlying database environments.

This past emphasis on reducing coupling between programs in general highlights commercial design as a primary influence of service-orientation, as further discussed in Chapters 8 and 9.

SUMMARY OF KEY POINTS

- A level of coupling is comparable to a level of dependency.
- Software coupling simply represents a connection or relationship between two programs or components.
- Many past architectural models established tightly coupled relationships among their parts.

7.2 Profiling this Principle

Service coupling is a multi-faceted, dynamic design characteristic that ties into and influences several other principles. On a fundamental level, this principle is concerned with the relationship between a service, its underlying environment, and its consumers (as previously illustrated in Figure 7.1).

As further detailed in Table 7.1, when trying to determine suitable levels of service coupling, our goal is to position the service as a continually useful and accessible resource while also protecting it and its consumers from forming any relationships that may constrain or inhibit them in the future (Figure 7.2).

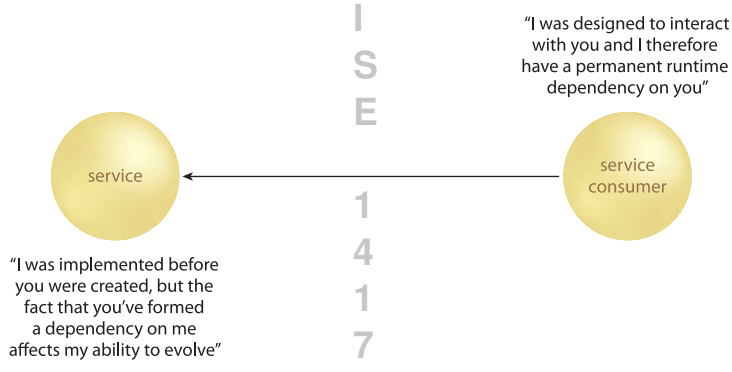

Figure 7.2
The fundamental impacts of coupling on the service and consumer. The constraints expressed in this figure are amplified when the measure of coupling and the quantity of consumers is increased.

NOTE

Because the scope of this chapter (and this book) is focused on service design, we are only concerned with coupling issues internal to the service architecture and between a service and its immediate consumers. There are various architectural design options for reducing coupling levels via the use of middleware and intermediaries. Some of these architectural coupling issues are also addressed by design patterns.

Principle Profile	
Short Definition	*"Services are loosely coupled."*
Long Definition	*"Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment."*
Goals	By consistently fostering reduced coupling within and between services we are working toward a state where service contracts increase independence from their implementations and services are increasingly independent from each other. This promotes an environment in which services and their consumers can be adaptively evolved over time with minimal impact on each other.
Design Characteristics	• The existence of a service contract that is ideally decoupled from technology and implementation details. • A functional service context that is not dependent on outside logic. • Minimal consumer coupling requirements.
Implementation Requirements	• Loosely coupled services are typically required to perform more runtime processing than if they were more tightly coupled. As a result, data exchange in general can consume more runtime resources, especially during concurrent access and high usage scenarios. • To achieve the right balance of coupling, while also supporting the other service-orientation principles that affect contract design, requires increased service contract design proficiency.

Web Service Region of Influence

As we explore different coupling types in the next section, it will become evident that applying this principle touches numerous parts of the typical Web service architecture. However, the primary focal point, both for internal and consumer-related design considerations, remains the service contract.

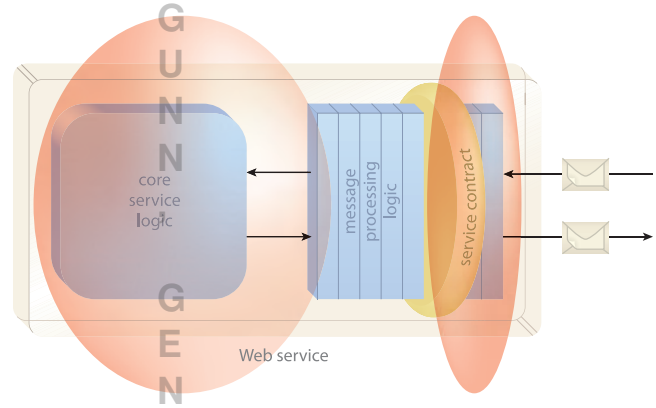

Figure 7.3

Table 7.1
A profile for the Service Loose Coupling principle.

SUMMARY OF KEY POINTS

- The scope of this principle affects both the design of a service, as well as its relationship with consumer programs.
- While we are concerned with how coupling affects service logic, the primary emphasis is on the design of the service contract.

7.3 Service Contract Coupling Types

To gain a better appreciation of the extent to which coupling can become part of service design, we need to take a look at the common types of relationships that need to be created within and outside of the service boundary.

This, the first of two sections that explores coupling types, is focused on dependencies that originate from within the service. How these relate to and influence the service contract directly ties into the subsequent *Service Consumer Coupling Types* section that

explores how less desirable forms of intra-service coupling can make their way into consumer program designs.

The service contract is the core element around which most coupling-related design considerations revolve. The basis of these issues is the relationship between the service contract and whatever logic and resources it encapsulates. As shown in Figure 7.4, we can design the contract with dependencies on the underlying service logic—or—we can choose to design the service logic with dependencies on the service contract.

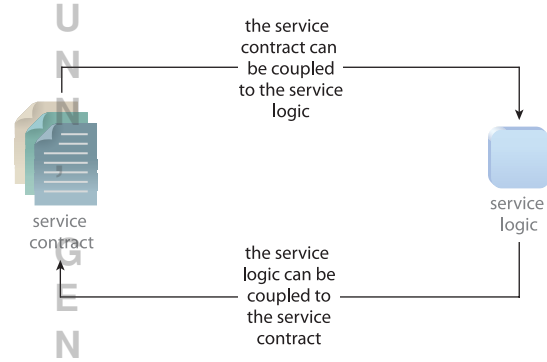

Figure 7.4
The illustrated forms of coupling are described in the upcoming *Contract-to-Logic Coupling* and *Logic-to-Contract Coupling* sections.

Understanding these two fundamental coupling types and the impact of proceeding with either is key to identifying the link between the design of individual services and the direction in which an entire service inventory will ultimately evolve. The design-time dependencies established between a service contract and its underlying logic have a direct bearing on the ultimate strategic potential of each service within a service inventory and, therefore, on the inventory as a whole.

Figure 7.5 positions the contract and its logic within the scope of common service-related dependencies that can exist as part of a typical SOA. This perspective also highlights the potential design complexity introduced by coupling-related issues.

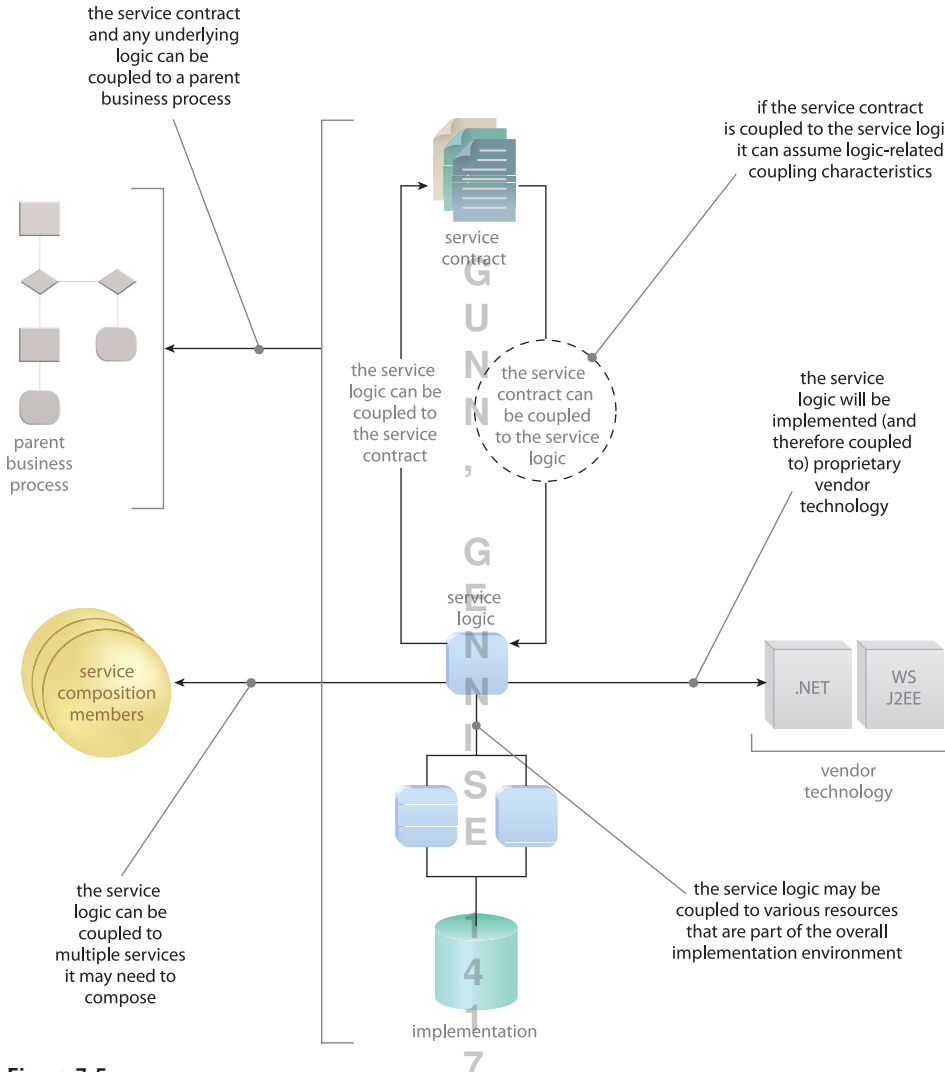

Figure 7.5

Coupling is a natural and unavoidable part of service architecture. It's knowing how and when to adjust the extent of coupling that gives us the ability to tune this architecture in support of service-orientation.

Various types of service-related coupling are represented in Figure 7.5, all of which can relate to both the service logic and contract. The same coupling types are re-displayed as part of a conceptual view in Figure 7.6.

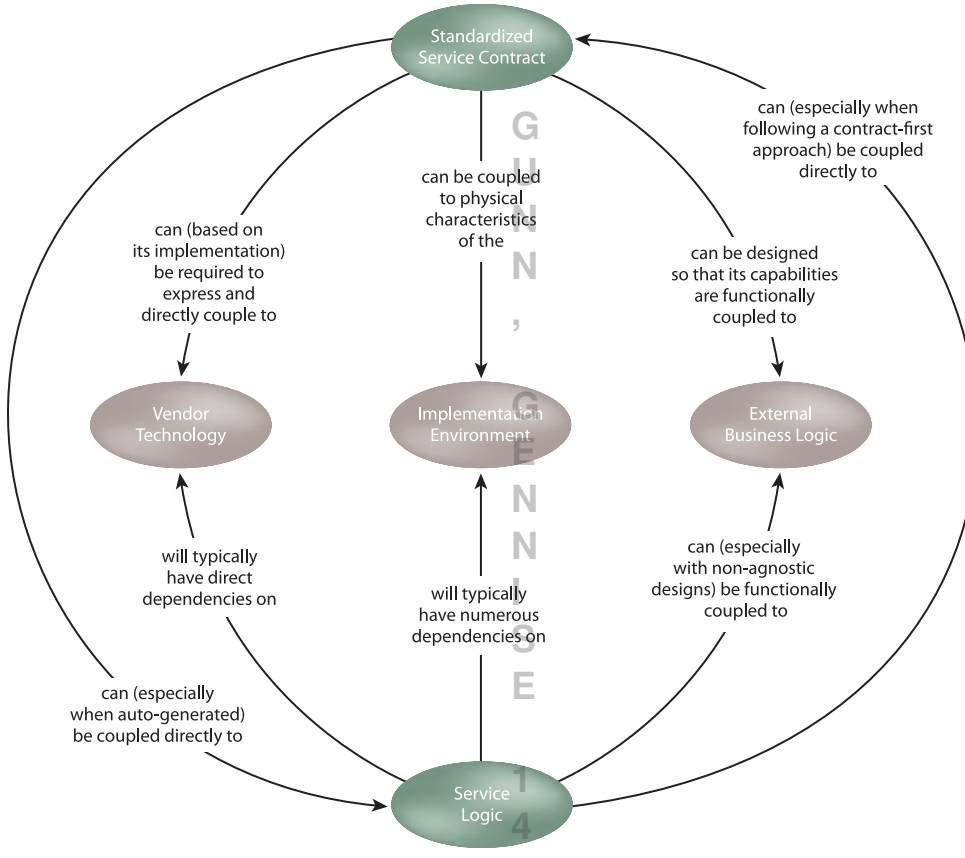

Figure 7.6
Both service contract and service logic can form dependencies on parts of the service environment and on each other.

From the relationships and dependencies depicted in Figures 7.5 and 7.6 we can extract a specific set of coupling types that are directly relevant to the design of services:

- Logic-to-Contract Coupling
- Contract-to-Logic Coupling
- Contract-to-Technology Coupling

- Contract-to-Implementation Coupling
- Contract-to-Functional Coupling

Except for logic-to-contract coupling, the goal behind this principle is to reduce the extent of all these coupling types. The following sections explore each type in more detail:

Logic-to-Contract Coupling (the coupling of service logic to the service contract)

A recommended approach to building a service is to design its physical contract prior to its underlying solution logic. This "contract first" process is very effective at ensuring that contract design standards are consistently incorporated. It also allows us to tune the underlying logic in support of the service contract, which can optimize runtime performance and reliability.

Following the contract-first process can result in the service logic being tightly coupled to the service contract (known as *logic-to-contract coupling*) because it is created specifically in support of the independently designed contract, as shown in Figure 7.7.

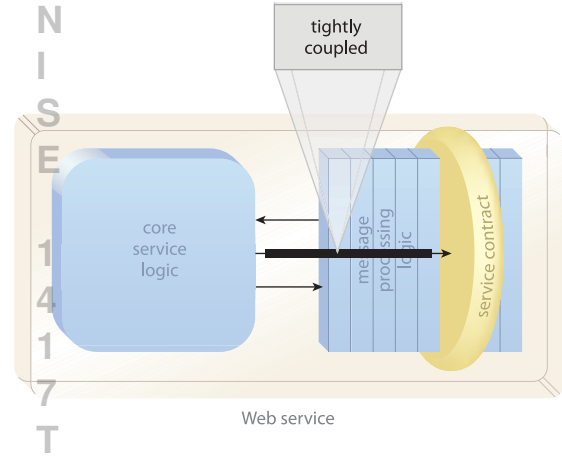

Figure 7.7
A Web service created through the contract-first process will naturally result in the service logic forming a tightly coupled relationship on the service contract. The contract, though, is not really coupled to the logic at all, allowing the service logic to be replaced in the future without affecting service consumers that have formed dependencies on the contract.

Despite the tight, unidirectional dependency formed by the logic on the contract, this is considered a positive type of coupling and a proven means of customizing services in support of service-orientation. It is further supported through the application of the Standardized Service Contract principle, which fundamentally preaches the contract-first design approach.

Of course, controlling the amount of required coupling between contract and logic is something we can only really accomplish when custom-developing services. In environments where service contracts need to be auto-generated or are provided by service adapter extensions, the level of coupling is often predetermined. In these cases there may be a need to resort to wrapper services, as further explored in the case study example at the end of this chapter.

FOR EXAMPLE

A multi-national pharmaceutical firm carried out a two-year SOA pilot program during which 47 Web services were created to automate 12 business processes within the human resources segment of the organization. A service design process was custom tailored for this project requiring that each Web service contract be fully developed prior to any further programming effort. Before the project could start, several of the programmers on the team had to undergo training in order to gain the required proficiency with WSDL and XML schema.

The end result was a moderately sized service inventory wherein 85% of services achieved high logic-to-contract coupling. Various capabilities within the remaining services were required to encapsulate legacy systems to one extent or another. This resulted in some service contracts having no logic-to-contract coupling at all (or having mixed levels of this coupling as implemented by a subset of the contract capabilities).

Note that in this project some legacy system encapsulation restrictions were overcome through the use of dedicated utility services that provided native translation of proprietary legacy APIs. (Specific design patterns exist to address legacy abstraction for service inventories.)

Contract-to-Logic Coupling (the coupling of the service contract to its logic)

The previous section described a coupling type based on the customization of the service contract prior to the development of the underlying service logic. This positions the contract as a relatively independent part of the service architecture, which maximizes the freedom with which the service can be evolved over time.

However, many contracts for Web services in particular have been derived from existing solution logic. This reverses the coupling dynamic, in that once these types of contracts come into existence, they find themselves immediately dependent on the underlying logic and implementation. As explained in Figure 7.8, this dependency of the contract upon its underlying logic is referred to as *contract-to-logic coupling*.

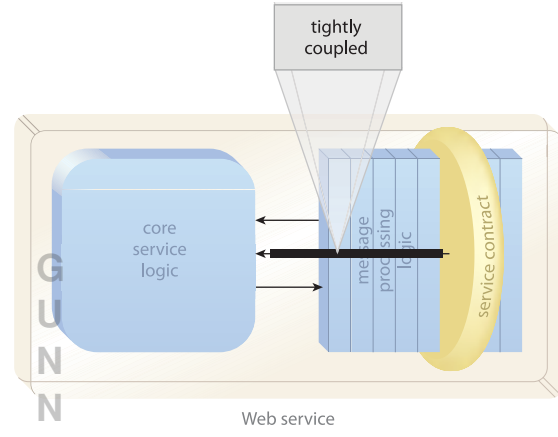

Figure 7.8
In environments where contracts can be auto-generated, the service logic is not dependent on the contract because if the logic changes, a new contract can be created at any time. The contract, however, is tightly coupled to the underlying service logic because the logic determines its design. Every time the logic changes and a new contract is generated, a new version of the service is effectively published, which raises numerous coupling and governance issues with consumers.

The most common examples have been the auto-generation of WSDL definitions using component interfaces as the basis for the contract design, as well as the auto-generation of XML schemas from database tables and other parts of physical data models. In both cases, the design of the resulting service contract is hardwired to the characteristics of its implementation environment. This is an established anti-pattern that shortens the lifespan of the service contract and inhibits the long-term evolution of the service.

NOTE

Services with contract-to-logic coupling will tend to have increased levels of technology, functional, and implementation coupling (as described in the upcoming sections).

FOR EXAMPLE

The aforementioned pharmaceutical company initiated the SOA pilot program to replace an integration architecture that had only existed for two years but had caused many problems. A legacy system acted as a hub for several custom-developed, distributed applications that exchanged data via five Web service endpoints.

A development tool had been used to auto-generate the source markup code that comprised each of the Web service contracts. As a result, the five WSDL definitions closely mirrored the five corresponding component interfaces. Each Web service contract therefore had a high level of contract-to-logic coupling. All of these Web services were eventually replaced.

Contract-to-Technology Coupling (the coupling of the service contract to its underlying technology)

A service that exists as a traditional proprietary component generally requires that the service contract be tightly coupled to the service's associated communications technology. As shown in Figure 7.9, the resulting *contract-to-technology* coupling may impose technology-specific characteristics on the contract as proprietary as the development technology used to build the service itself.

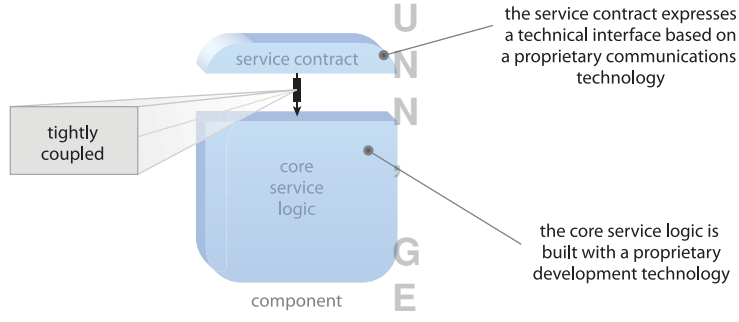

Figure 7.9
A service developed as a proprietary component can require that the service contract exist as a proprietary extension of the service. This couples the contract to the implementation technology which, in turn, imposes the requirement that all service consumers support the same proprietary (or non-industry standard) communications protocol.

As further explained in the *Consumer-to-Contract Coupling* section of this chapter, making the technical service contract dependent on proprietary technology limits the potential consumers to those who are capable of supporting the technology.

The ability to abstract proprietary technology in support of a non-proprietary framework is what has made Web services so successful. A Web service contract is not required to express proprietary details of the underlying solution logic and can be positioned to exist as an independent part of the service architecture. This not only frees consumer programs from having to comply with proprietary communication protocols, it also gives the service owner the ability to swap service implementation technologies without affecting the service's existing consumer base.

FOR EXAMPLE

In its initial incarnation, the pharmaceutical company's human resource integration environment consisted of .NET components only. Data sharing was effectively accomplished as long as both ends of every integration channel supported the .NET Remoting protocol they had standardized on at the time. Each of the components involved therefore exposed technical interfaces with contract-to-technology coupling.

As interoperability demands increased, the existing architecture was deemed too restrictive because it limited interaction to programs capable of supporting .NET Remoting. The previously mentioned Web services were introduced and strategically positioned to allow for messaging-based data exchange over HTTP. These Web service endpoints overcame the contract-to-technology coupling limitations by exposing vendor technology-neutral contracts.

The subsequent SOA program that replaced these Web services introduced new, standardized Web services to continue to avoid contract-to-technology coupling.

Contract-to-Implementation Coupling (the coupling of the service contract to its implementation environment)

Any physically deployed service will be comprised of or will require access to a collection of implementation technologies and products beyond the core service logic.

Examples include:

- physical databases and associated physical data models
- legacy system APIs
- user and group accounts and associated physical directory structures
- physical server environments and associated domains
- file names and network paths

It is relatively normal for some forms of service logic to be bound and connected to these details. This enables the logic to effectively access and interact with these resources as required at runtime.

When deriving service contracts from logic bound to an implementation environment, implementation-specific characteristics and details can become embedded within the contract content. In the case of XML, the auto-generation of the schema from database tables or views will similarly end up placing physical data model details into the service contract. The result is a direct dependency formed by the service contract on the underlying implementation, referred to as *contract-to-implementation coupling* (Figure 7.10).

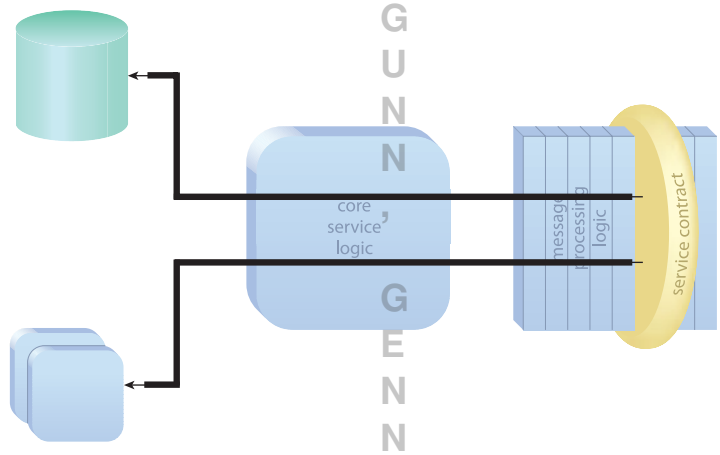

Figure 7.10
A Web service consisting of service logic, external components, and a database, the latter two of which impose implementation-specific details onto the service contract content.

Note that the manner in which the service logic itself relates to its underlying implementation can also establish a related form of coupling called *logic-to-implementation coupling*. As we will learn in Chapter 10, it is highly preferable for the implementation resources required by a service to be dedicated. However, almost every service architecture is unique, and desired levels of autonomy are not always possible. Therefore, we need to account for situations where the parts of the service logic that are dedicated are required to access (and therefore form dependencies on) parts of the enterprise that exist external to the service boundary.

The extent to which the service relies on external resources determines the level of coupling the service forms on its surrounding implementation environment. There are concrete benefits to minimizing this form of coupling, all of which are addressed by the Service Autonomy principle.

FOR EXAMPLE

A corporation owning several lumber mills had a centralized IT department that custom developed a handful of applications to manage some of the more unique aspects of their business. Subsequent integration requirements prompted IT managers to explore XML as a standard data representation format. For each data exchange requirement a utility was used to derive one or more XML schemas from existing database tables and views.

This approach fulfilled immediate requirements but caused some challenges when several Web services were later introduced to accommodate a new business process that imposed changes on the integration architecture. The schemas became part of the Web service contracts. They introduced an extent of contract-to-implementation coupling because their complex types were directly derived from composites of table columns and fields.

Here is a sampling of one of the complex types containing embedded table column names and the questionable use of an attribute:

```
<xsd:element name="BZN_TAB22">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="BZN_DET_SHP"
          type="xsd:string"/>
      <xsd:element name="BZN_TS_DAT67"
          type="xsd:base64Binary"/>
    </xsd:sequence>
    <xsd:attribute name="BZN_A_EMP_NAME"
        type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:element>
```

NOTE

Service contracts can include WS-Policy definitions capable of expressing a variety of policy assertions. Some of these assertions may be comprised of syntax or characteristics proprietary to the vendor platform used to generate the definition or perhaps proprietary to business rules and polices predefined within vendor products or legacy systems. These cases could also be classified as forms of contract-to-implementation coupling.

Contract-to-Functional Coupling (the coupling of the service contract to external logic)

When the logic encapsulated by a service (or, more specifically, by one of its capabilities) is specifically designed in support of a body of functionality that exists outside of the service boundary, then the corresponding service contract can become functionally coupled, resulting in *contract-to-functional coupling*.

There can be many variations of functional coupling. Provided here are some common examples:

Parent Process Coupling

Functional coupling can exist between the logic encapsulated by service capabilities and business process logic represented and implemented elsewhere in the enterprise. If a service has been designed specifically to support a particular business process, its logic as well as its contract might well be tightly coupled to the logic of that process.

Service-to-Consumer Coupling

A service can be designed to support a particular (usually pre-existing) service consumer program. This is common in B2B architectures, where an established organization already has a service consumer in place. Partners wishing to participate online are required to deliver services according to design standards dictated by the organization and their environment. Another typical occurrence of consumer coupled services is within internal point-to-point integration architectures, where both service and consumer are built only to work with each other to establish a specific integration channel.

Either way, intentionally designing a service for a single consumer (or for a limited amount of consumers) typically results in consumer-specific functional coupling. Note that this may or may not also result in parent process coupling, depending on the nature of the functionality being delivered by the service.

Functional Coupling and Task Services

In the case of a task service, we are deliberately limiting the functional scope to that of a business process. We generally do so with the assumption that the service encompasses the scope of the process and therefore acts as the parent controller. Other services avoid this type of coupling by basing their functional scope on an agnostic context (such as a business entity). A task service can be considered an example of intentional or *targeted* functional coupling.

NOTE

Focusing on logic-to-contract coupling and avoiding the other negative forms of coupling described in this section leads to increased design-time control of a service, as explained in the *Design-Time Autonomy* section of Chapter 10.

SUMMARY OF KEY POINTS

- There are many forms of coupling that relate to internal and external service design and runtime processing. All represent relationships and dependencies that exist between different architectural components.
- Numerous coupling-related concerns revolve around the service contract. If the contract for a service can be standardized, many undesirable types of coupling between the contract and its underlying implementation can be avoided.

7.4 Service Consumer Coupling Types

Ultimately, it is the service and one of its consumers that will need to interact to carry out some form of business task. How their relationship is defined at design-time determines the level of cross-service coupling they will need to live with, as explained in Figure 7.11. This relationship is therefore a core design consideration.

Two very specific types of consumer coupling are explored in this section:

- Consumer-to-Implementation Coupling
- Consumer-to-Contract Coupling

The primary distinction between these types is whether or not the service contract is accessed as the sole or primary endpoint into service logic and resources.

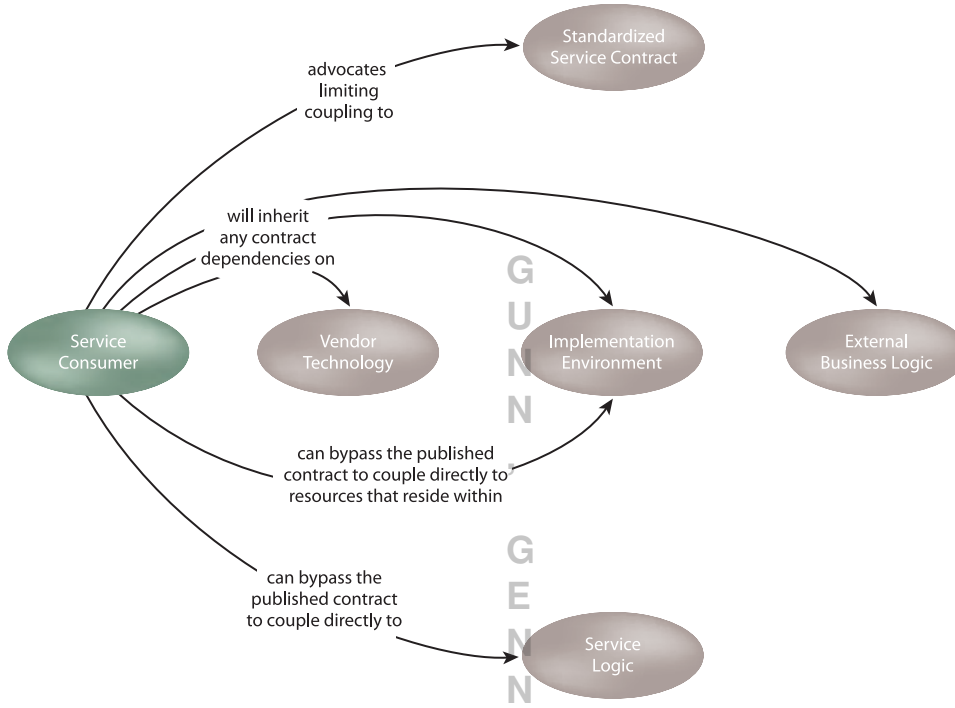

Figure 7.11
Consumer programs can form various types of dependencies on service resources, either via the service contract or by circumventing it.

Consumer-to-Implementation Coupling

A service consumer is technically not forced to access a service via its contract, as evidenced by Figure 7.12. There are often other entry points that may seem more attractive for reasons such as improved performance and design simplicity. However, these result in undesirable forms of *consumer-to-implementation coupling* that can inhibit both service and consumer in the future.

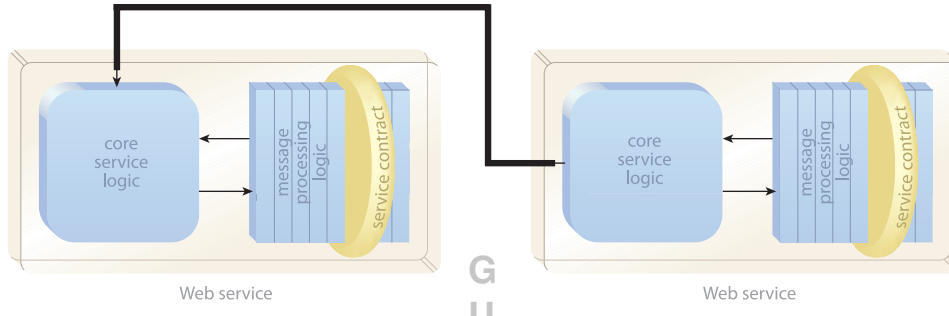

Figure 7.12
The service consumer bypasses the published service contract and accesses (and tightly couples to) the underlying service logic directly.

The first question a consumer program designer needs to answer is whether a service's published contract will be used at all. When designing a program to access or use a resource or capability that belongs within the boundary of a service, there are usually several options that exist as to how the consumer's data sharing requirements can be fulfilled.

Many of these options are reminiscent of past integration architectures, where often performance and ease of connectivity helped determine the most suitable integration channel between two applications. As illustrated in Figure 7.13, consumer programs can be designed with this approach in mind, leading them to disregard the service contract and connect directly to underlying resources.

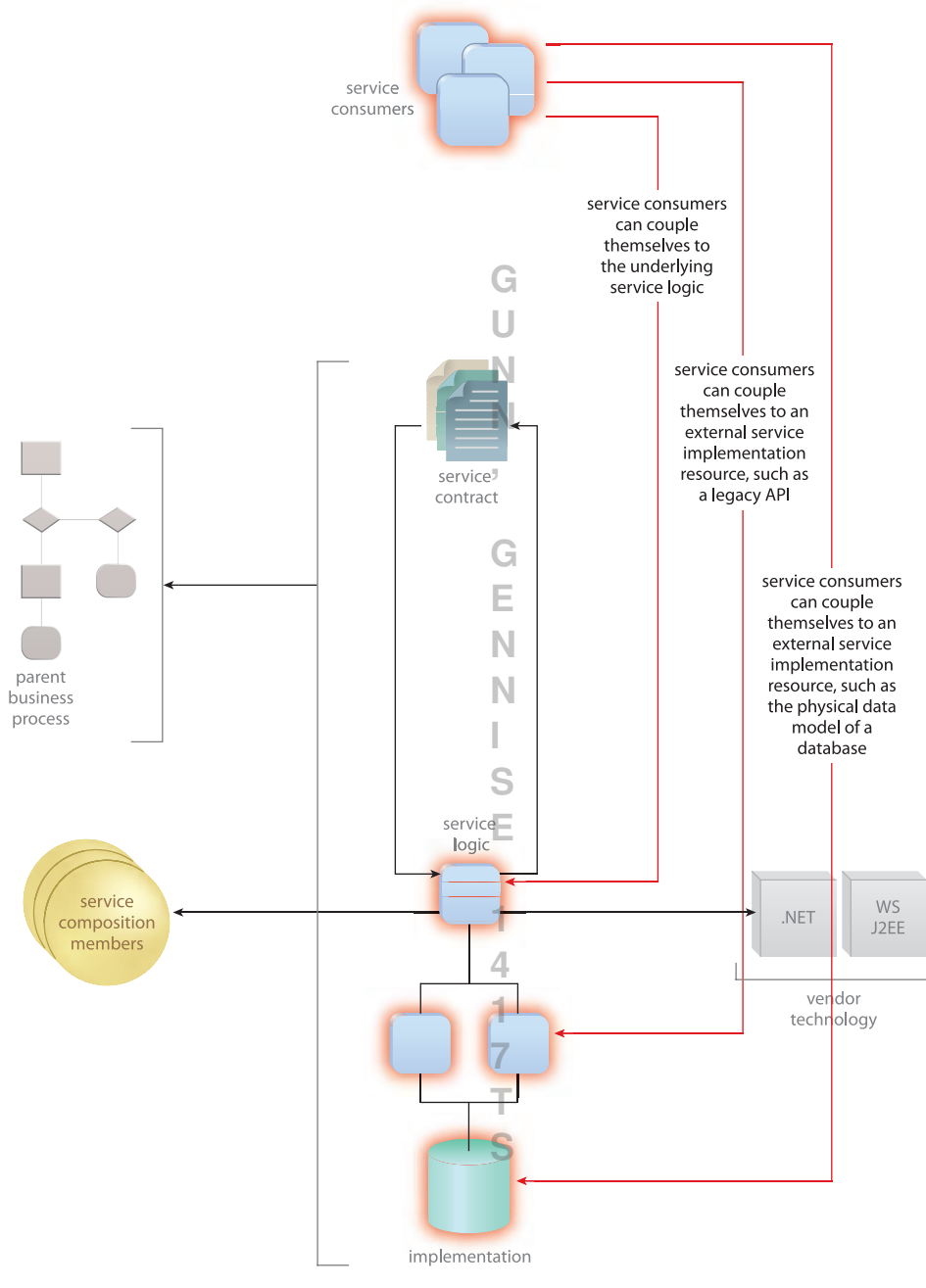

Figure 7.13

Consumer programs are designed to disregard the service contract and access underlying resources directly. While this may lead to more efficient data sharing channels, it is in fact an anti-pattern that severely undermines the goals of service-orientation.

Standardized Service Coupling and Contract Centralization

To address the pitfalls associated with bypassing the service contract, a standards-related design pattern known as Contract Centralization provides a simple solution for effectively and consistently implementing the appropriate form of consumer coupling.

Centralization simply means limiting the options of something to one. From a consumer's perspective, this pattern simply asks us to restrict access to a service to its contract only (thereby reducing or eliminating consumer-to-implementation coupling). Consumer programs either adhere to centralization, or they don't. If they do, then they only establish connections as described in the upcoming *Consumer-to-Contract Coupling* section.

NOTE

The centralization of service contracts is a standards-based concept also relevant to furthering the application of the Service Reusability principle. As explained in the *Standardized Service Reuse and Logic Centralization* section in Chapter 9, the associated Logic Centralization design pattern requires that certain bodies of logic be accessed only through certain (centralized) services. Contract Centralization and Logic Centralization therefore can be positioned as cornerstone enterprise standards that directly support SOA.

Consumer-to-Contract Coupling

Regardless of whether a service contract is fully centralized, any time a consumer binds to its contract, the resulting relationship can simply be referred to as *consumer-to-contract coupling* (Figure 7.14).

This is a recommended and desirable form of coupling because it achieves the greatest amount of independence between the consumer and the service. Consumer-to-contract coupling essentially forms the basis of a loosely coupled cross-service relationship, as promoted by this principle. However, the extent of "coupling looseness" actually attained is determined by the content of the service contract.

All variations of coupling we covered in the *Service Contract Coupling Types* section are relevant to consumer-to-contract coupling because the consumer program is required to physically bind to a capability expressed as part of the technical service contract. As a result, it will end up forming a dependency on anything to which that part of the service contract is coupled.

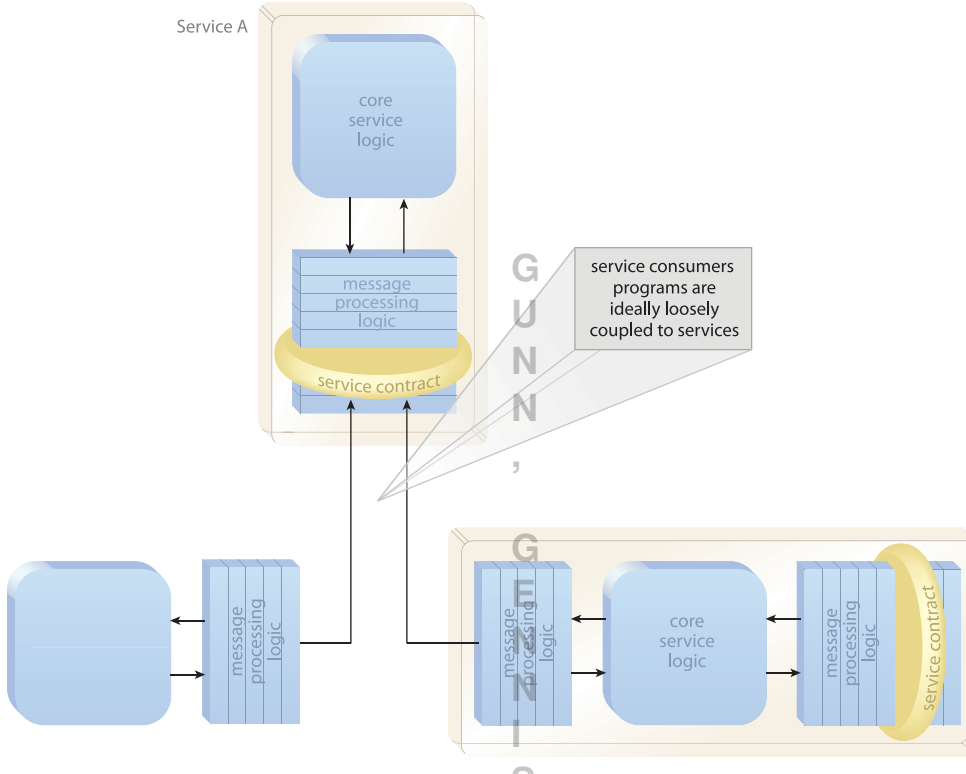

Figure 7.14
In this scenario, a Web service (bottom right) and a regular component (bottom left) are both designed to work with Service A (a Web service). The contract published by Service A was designed specifically to minimize consumer dependencies, resulting in loosely coupled consumer relationships.

This form of "coupling inheritance" is a constant concern, especially with agnostic services, because we want to avoid the proliferation of undesirable coupling characteristics throughout multiple service consumers, as highlighted in Figure 7.15.

Direct and Indirect Coupling Scenarios

A service contract tightly coupled to other parts of the service architecture will find itself expressing (usually physical) details about its underlying implementation. As we've established, this has a domino effect in that all subsequent service consumer programs that end up forming dependencies on the service contract also become coupled to the very same implementation characteristics.

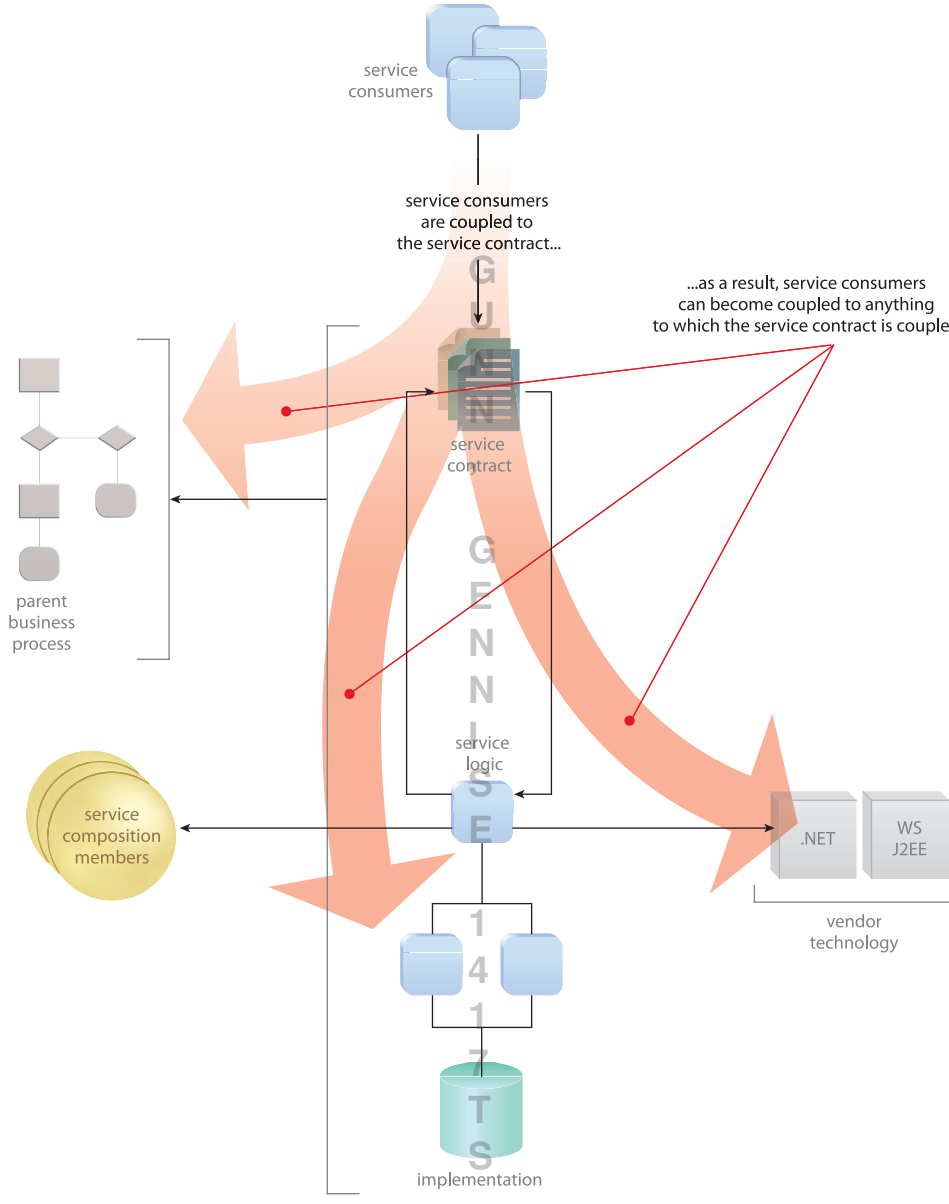

Figure 7.15
Service consumers inherit undesirable coupling characteristics embedded within the service contract, which can lead to the consumer programs forming dependencies on the underlying service environment.

The next set of figures revisits three of the previously explained service contract coupling types to demonstrate how each can result in *direct* or *indirect* consumer coupling.

Figure 7.16 shows that when the service contract is technology-coupled, its consumers will likewise become technology-coupled. This is a direct form of negative coupling because the consumer designer is fully aware of the coupling-related technology requirements during design-time.

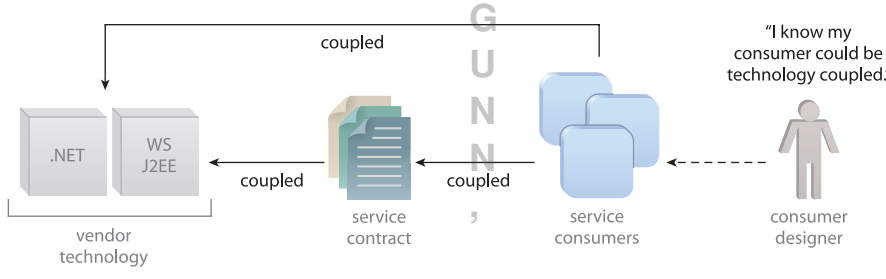

Figure 7.16

When the service contract is technology coupled, its service consumers will be forced to become coupled to the service's underlying technology.

Figure 7.17 illustrates that if the service logic is functionally coupled to external business logic AND if the service contract is coupled to the same service logic, then the resulting contract-to-functional coupling can be imposed on consumers. This can be an indirect form of negative coupling because the consumer designer may not know of the service's dependency on a parent business process.

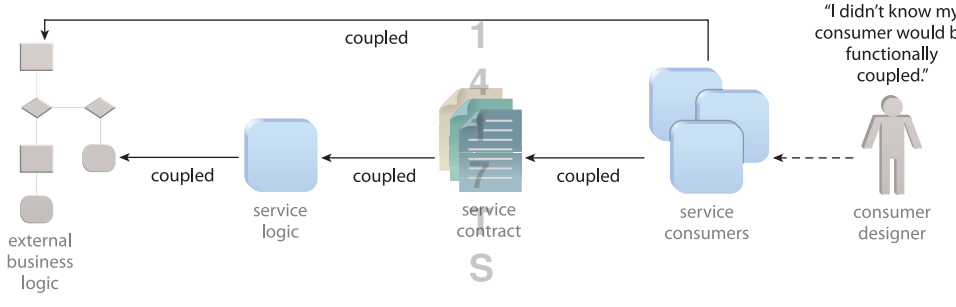

Figure 7.17

When the service contract is functionally coupled, its service consumers are required to couple to the service's underlying functional dependencies.

Figure 7.18 further proves that if the service logic is coupled to implementation resources AND if the service contract is (partially or entirely) derived from these resources, then the resulting contract-to-implementation coupling can lead to further proliferation of implementation coupling by service consumers. This is perhaps the most common form of indirect negative coupling.

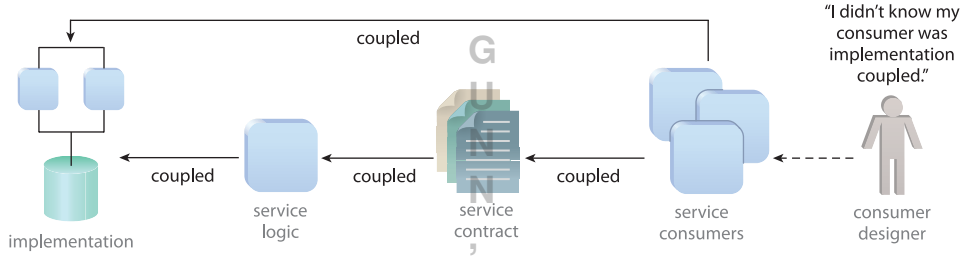

Figure 7.18

When the service contract is derived from parts of the service's implementation resources, its consumers will also become coupled to those parts of the implementation environment. This is especially undesirable when the resources do not belong exclusively to the service but are instead shared parts of the overall architecture.

One of the greatest challenges to avoiding indirect coupling is that, due to deliberate information hiding policies that result from the application of the Service Abstraction principle, many service consumer designers may be completely unaware of the fact that their programs are in fact being (indirectly) coupled to underlying service details. It is therefore the responsibility of the service designers to minimize negative forms of contract coupling in the first place.

The level of dependency we establish between individual, physically separate services can have profound implications as to how effective a service inventory can become in support of future service composition requirements. We therefore need to pay close attention to the extent of coupling a service demands of its consumers.

Contract Centralization and Technology Coupling

When standardizing on the service contract as the sole service endpoint, we are forcing all consumers to comply with the interaction requirements expressed by that contract. If the contract technology is proprietary or requires the use of proprietary communication protocols, then we limit the consumer base to those programs compatible with the proprietary requirements (Figure 7.19).

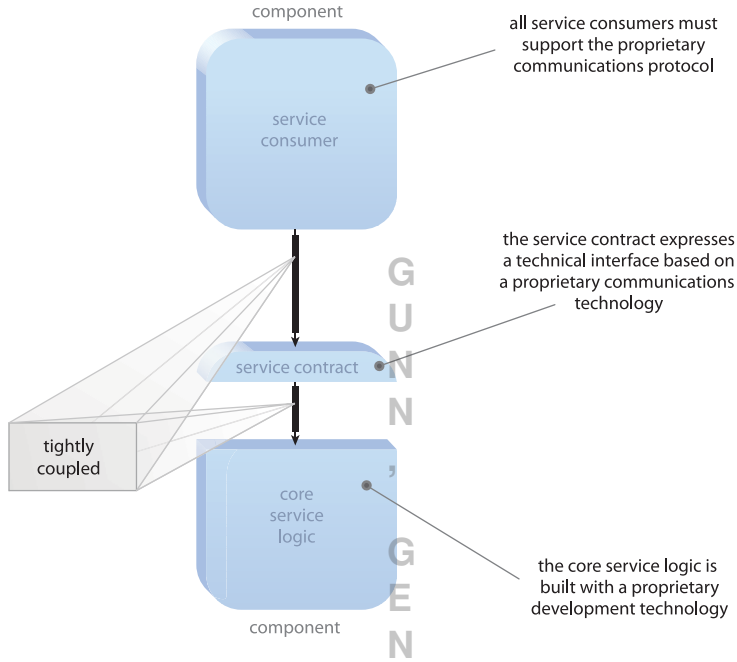

Figure 7.19
The figure originally displayed in the *Contract-to-Technology Coupling* section is revisited to show how the tight technology coupling of the service contract is passed on to a service consumer.

If the centralization of contracts is enforced to a meaningful extent, we make the service contract a focal point for a great deal of interaction. From a long-term evolutionary perspective, therefore, Web services provide an effective means of establishing a service contract that can be customized and standardized, while remaining decoupled from the service's underlying technology.

Without the use of an open technology platform, such as Web services, Contract Centralization can result in the proliferation of technology coupling throughout an enterprise.

Validation Coupling Considerations

Regardless of the extent of indirect coupling a service contract imposes, there will always be the requirement for the consumer program to comply to the data model defined in the technical service contract definitions.

In the case of a Web service, this form of *validation coupling* refers to the XML schema complex types that represent individual incoming and outgoing messages. Schemas establish data types, constraints, and validation rules based on the size and complexity of the information being exchanged as well as the validation requirements of the service itself.

The extent of validation coupling required by each individual service capability can vary dramatically and is often tied directly to the measure of constraint granularity of service capabilities. As per the validation-related design patterns, each contract can be individually assessed as to the quantity of actual constraints required to increase its longevity.

Consumer Coupling and Service Compositions

What happens when a service composes another that has a particular level of coupling established with a third service? Are negative coupling characteristics (such as those related to the implementation) passed on from the third service through to the first? These types of issues will arise when designing service compositions. Chapter 13 explores service composition, and the corresponding Service Composability principle addresses some of the concerns associated with cross-service relationships. However, inter-service coupling measures still deserve individual attention.

NOTE

The case study at the end of this chapter documents the scenario just described and further addresses commonly raised questions.

Measuring Consumer Coupling

Because of the uniqueness of service capabilities and consumer requirements, each interaction between a consumer and a service capability will be distinct. It is therefore helpful to establish a set of categories that we can use to represent measures of consumer coupling.

Based on the possible variations of consumer coupling that are technically possible, there are many classifications one could come up with to label different coupling levels. Because the Contract Centralization pattern is so fundamental to establishing the most beneficial forms of consumer coupling, we can define two basic levels that address the following questions:

- Is the coupling centralized?
- If it is, what is the degree of required contract coupling?

The following generic categories define corresponding coupling levels and provide a means of communicating the coupling requirements of individual service capabilities.

Non-Centralized Consumer Coupling

The body of logic represented by the service is not accessed solely by consumer programs via the service contract. The actual coupling requirements are therefore dependent on the individual access points chosen by the consumer.

Centralized Consumer Coupling

To measure the level of a centralized coupling relationship requires the identification and assessment of each of the coupling types that can affect the content and coupling requirements of the service contract.

Therefore, one approach for documenting coupling levels is to create a profile for every service capability in which the dependency level of each of the previously described coupling types is assessed. A numeric rating system can be used (ranging from 1 to 5, for example), or standard classification terms (such as "low," "moderate," and "high") can be applied.

BEST PRACTICE

The environment in which you are required to create Web services will often dictate constraints that make some negative forms of coupling unavoidable. Wherever possible, apply this principle to reduce the *extent* of these coupling types prior to implementing the service for production use. At a minimum, this principle can be used to establish an awareness of negative coupling types associated with a service so that service consumer designers are fully cognizant of how these intra-service dependencies may impact their use of the service over the long-term.

Specifically, coupling levels can be documented as part of a service profile for communication purposes during service design processes. These levels can be furthermore made available as part of the overall service contract to openly reveal negative forms of coupling to whatever extent allowed when taking Service Abstraction considerations into account.

SUMMARY OF KEY POINTS

- The key motivation for decoupling a service contract from its implementation is to avoid having service consumers indirectly couple to service implementation details.
- The Contract Centralization pattern requires that consumers interface only with the official service contract and not with other potentially available service entry points.
- Consumer-to-contract coupling is an approach used to avoid consumer-to-implementation coupling. However, when based on a poorly designed service contract, consumer-to-contract coupling can still result in the consumer becoming coupled to the service implementation.

7.5 Service Loose Coupling and Service Design

The following sections position previously discussed coupling types and Contract Centralization within the service-orientation paradigm as a whole and also take a look at how service coupling affects the design of individual service models.

Coupling and Service-Orientation

We've defined an array of coupling types and discussed how they relate to the notion of Contract Centralization. When we tie all this together we can see how, through this principle and others, service-orientation promotes loose coupling within and between services.

As mentioned in point number 2 in Figure 7.20, other service-orientation principles get involved in shaping the structure and content of service contracts. Table 7.2 explores this further by listing coupling types along with associated principles.

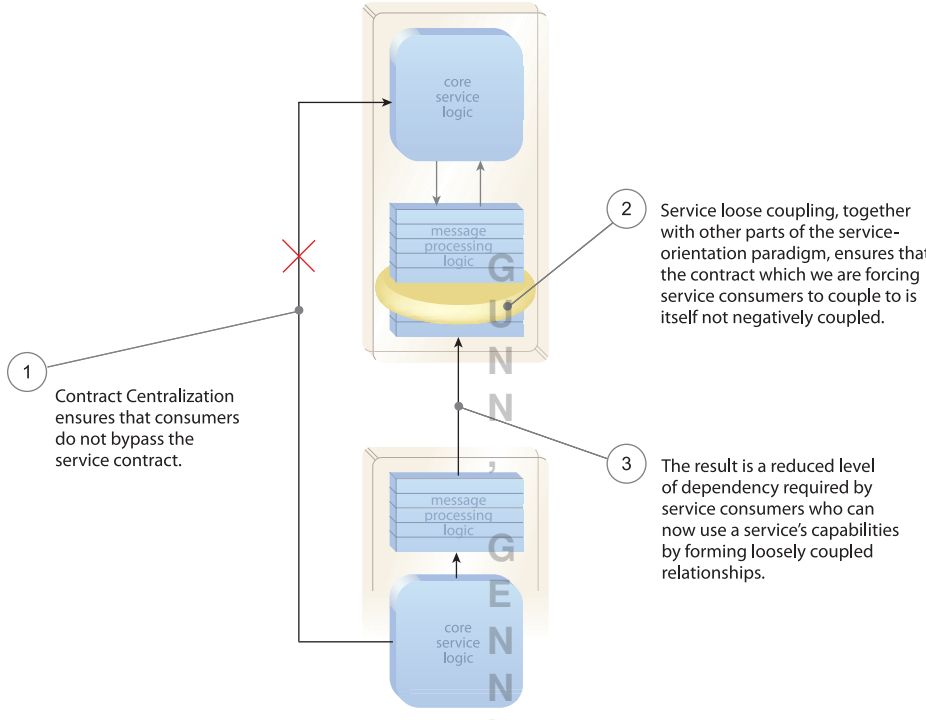

Figure 7.20

Through Contract Centralization we place the service contract front and center within a service-oriented architecture. This is why much of service-orientation is focused on contract design.

Coupling Type	Negative?	Note
Logic-to-Contract	No	Tight coupling of the service logic to the contract is acceptable and supported by the Standardized Service Contract principle.
Contract-to-Logic	Yes	This form of coupling is not recommended and can be avoided through the use of "contract first" design approaches.
Contract-to-Technology	Yes	The service contract is ideally decoupled from vendor technology, as supported by the use of open XML and Web services standards.

Coupling Type	Negative?	Note
Contract-to-Functional	Yes	This negative coupling type is avoidable through the application of the Service Reusability principle but may still be required for certain types of services.
Contract-to-Implementation	Yes	This form of coupling is not recommended, especially in relation to external and shared implementation resources.
Consumer-to-Implementation	Yes	The Contract Centralization design pattern is used specifically to avoid this coupling type.
Consumer-to-Contract	No	This is a positive form of coupling, but its benefit is related to the extent to which negative service contract coupling levels have been avoided.

Table 7.2
A summary of coupling types and associated influences.

Service Loose Coupling and Granularity

Granularity concerns associated with this principle primarily relate to consumer-to-contract coupling, which can affect capability, data, and constraint granularity levels.

The functional scope of a capability exposed by a service contract is what the service consumer is required to commit to when forming a design-time dependency on the contract. If the capability performs too much work, the consumer may be negatively affected by the extra processing overhead. Alternatively, if the functional scope of the capability is too fine-grained, the consumer may be negatively affected by increased service roundtrips (either by having to call the same capability repeatedly or by having to call additional capabilities).

Consumers are further required to submit to whatever level a capability's data granularity is set at. If the required data exchanges are too fine-grained, the consumer may not receive sufficient information and may be required to invoke additional services. And on the flipside, if the data is too coarse, the consumer may receive more information than it actually needs, which can waste bandwidth and consumer processing cycles.

Constraint granularity can also play a role in determining coupling requirements. Larger amounts of fine-grained constraints can increase the amount of validation logic consumers are required to comply with (as also discussed in the aforementioned *Validation Coupling* section).

Clearly, determining the appropriate granularity levels is important for a service to be effectively utilized, especially when providing reusable functionality. There is no one level that is perfect for all possible consumers. In fact, the Contract Denormalization pattern advocates providing similar capabilities with different granularity levels to accommodate different types of consumers.

Fundamentally, though, it is the service contract designer's understanding of how granularity directly influences consumer coupling requirements that helps determine the right balance of capability, data, and constraint granularity.

Coupling and Service Models

Because there are so many types of dependencies that can exist to varying extents, the actual coupling that results really comes down to the nature of the service logic as well as the manner in which it is delivered and then implemented, regardless of service model. However, there are some coupling-related tendencies associated with service models worth mentioning:

Entity Services

Entity services are generally delivered as part of a service inventory modeling project supplemented with design standards required to establish the entity service layer. They are ideally custom designed to make the most of top-down analysis efforts required to properly model service boundaries. They therefore present an ideal opportunity to create services that avoid many of the negative coupling types by establishing highly independent (and decoupled) service contracts.

The only external part of the enterprise to which entity services are tightly coupled is the business entities themselves. Therefore, if an organization's fundamental lines of business change, so too can the complexion of its information architecture and associated enterprise entity model. However, in most cases, business entities provide a safe functional context that is business-centric while remaining agnostic to multiple business processes. In fact, it is the increased longevity of business entities within the lifespan of an organization that makes the entity service model so attractive.

Utility Services

Because utility services are often required to encapsulate existing enterprise resources (including legacy systems), they can easily become implementation-coupled. In this case, it is important to design a standardized service contract whenever possible to avoid indirect consumer coupling to the implementation as well.

Utility services are also frequently developed as components using native vendor technology. Often this approach is taken out of necessity for performance and reliability reasons when the surrounding vendor platform's support for Web services technology is deemed insufficient. This will result in technology coupling requirements until these services can be wrapped with and exposed via open Web service contracts.

Task Services

Due to their specific functional context, task services are sometimes functionally coupled. If the business process logic encapsulated by the service represents a sub-process of a larger parent process, then the service logic will be directly dependent on that external business logic.

Furthermore, task services required to perform unique (and especially smaller scoped) activities are sometimes created for single clients, making them service-to-consumer coupled.

Orchestrated Task Services

Because orchestrated task services rely on the deployment environment provided by a vendor orchestration platform, there is a natural dependency between the solution logic and its implementation.

The use of open Web services standards, such as WS-BPEL, can (to a large extent) avoid technology coupling. However, because some of these standards require that the Web service contract be appended with orchestration-specific constructs, a level of contract implementation coupling may be unavoidable.

How Service Loose Coupling Affects Other Principles

With such an emphasis on how dependencies relate to and originate from within service contracts, the application of this principle naturally affects other principles also concerned with service contract design or influenced by reduced coupling levels (Figure 7.21).

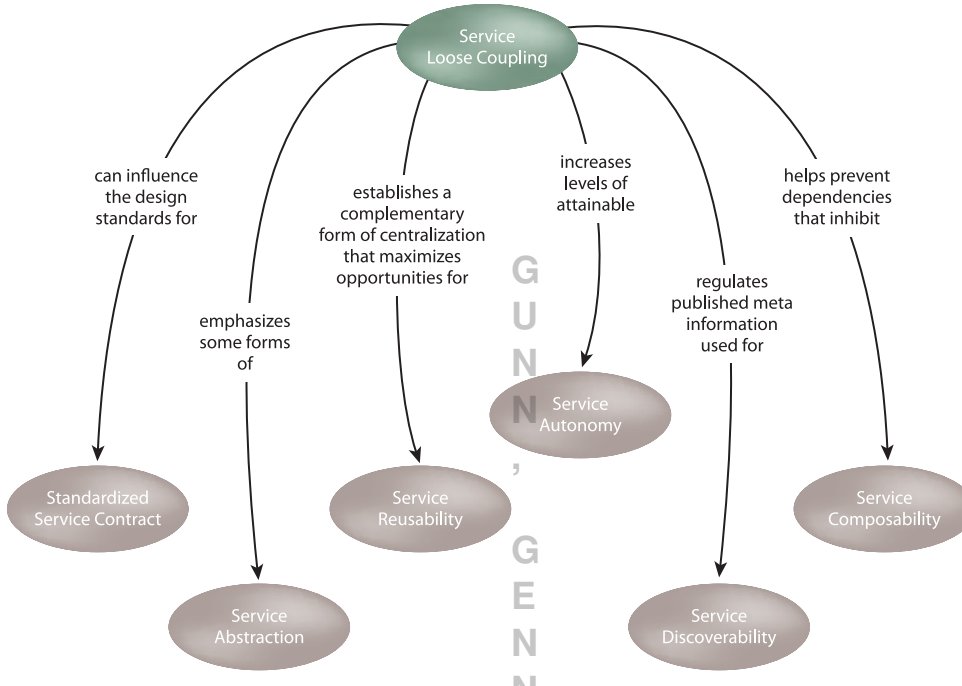

Figure 7.21

Service coupling emphasizes a reduction of internal and external service dependencies, which ends up supporting and affecting various aspects of other principles.

Service Loose Coupling and Standardized Service Contract

Establishing consistently standardized service contracts requires the existence and use of contract design standards. Often these standards will be rigid with many requirements related to schema structure, data types, validation constraints, and business rules. Loose coupling encourages us to moderate the quantity and complexity of technical contract content so as to minimize consumer dependency requirements and maximize the freedom service owners can have to evolve and change the service over time without affecting existing consumers.

Service Loose Coupling and Service Abstraction

As established in Chapter 5, Service Loose Coupling and Service Abstraction go hand-in-hand. The emphasis on creating less coupled consumer relationships specifically requires well-defined levels of functional and technology abstraction to be applied (as explained in Chapter 8).

Service Loose Coupling and Service Reusability

Decreasing dependencies allows us to do more with services in the long-term. They can be more easily composed, evolved, and even augmented in support of changing business requirements and directions. The ability to effectively reutilize and repurpose existing services is what reuse is all about. The enablement of loosely coupled relationships within a service and across a service inventory maximizes the potential for leveraging enterprise-wide reuse opportunities.

Service Loose Coupling and Service Autonomy

Reduced levels of negative coupling types support the realization of higher runtime and design-time autonomy levels. There is also a direct correlation between consumer coupling levels and service autonomy in that the more cross-service dependencies a service consumer has, the less autonomous it can become.

Service Loose Coupling and Service Discoverability

Because Service Discoverability is concerned with making services easily located and understood, it tends to encourage us to outfit service contracts with meta rich content. The Service Loose Coupling principle can sometimes push back at this requirement and help regulate the amount of contract content to what is actually necessary. Note that unlike Service Abstraction, Service Loose Coupling is primarily concerned with technical contract content or any part of the published meta information that would allow a consumer program to form a direct dependency.

Service Loose Coupling and Service Composability

Negative forms of coupling within a composed service can have a direct impact on the larger composition, as follows:

- *Contract-to-Logic Coupling*—If the service contract is auto-generated, it will likely not conform to standards in use by other services, therefore resulting in the need for transformation between it and other composition members.
- *Contract-to-Technology Coupling*—If a combination of open and proprietary service technologies are in use as part of the same composition, native technology conversion layers might be required. For example, Web services can compose services that exist as components; however, if those components then compose services that exist as Web services, data exchanges need to undergo two levels of technology transformation.

- *Contract-to-Implementation Coupling*—When a service contract is coupled to underlying implementation characteristics, it ends up imposing those characteristics upon the composition as a whole.

Using the previously described coupling levels to communicate the nature and extent of service dependencies can therefore be highly beneficial when modeling service compositions.

SUMMARY OF KEY POINTS

- Service Loose Coupling helps shape the application of other principles because much of general service design ties into or affects some form of coupling.
- This principle introduces concepts and considerations that go beyond the scope of other principles.

7.6 Risks Associated with Service Loose Coupling

There are obvious risks that come with allowing negative coupling types within a service contract. For example, it is evident that long-term issues will arise when a technology or implementation-coupled service contract passes on these coupling requirements to all of its consumer programs. However, even when pursing the loosely coupled ideal, there are additional risk factors that deserve to be taken into account.

Limitations of Logic-to-Contract Coupling

The contract first approach is a recommended process for delivering services that leads to tight logic-to-contract coupling because service logic is built after and tailored for the customized service contract.

However, a design concern raised by this approach is the limitation of having just one service contract associated with the core service logic. There are times when it is preferable to have two or more contracts for the same underlying logic so as to establish multiple points of entry, each exposing different service capabilities for different types of consumers.

As shown in Figure 7.22, this type of service design requires us to establish a lower degree of coupling between logic and its contract. By taking multiple potential service

contracts into account early in the design and development of the core logic, the risk of limiting the service to one contract can be mitigated. The use of the Service Façade design pattern accommodates this issue by introducing a separate layer of abstraction within the underlying service logic.

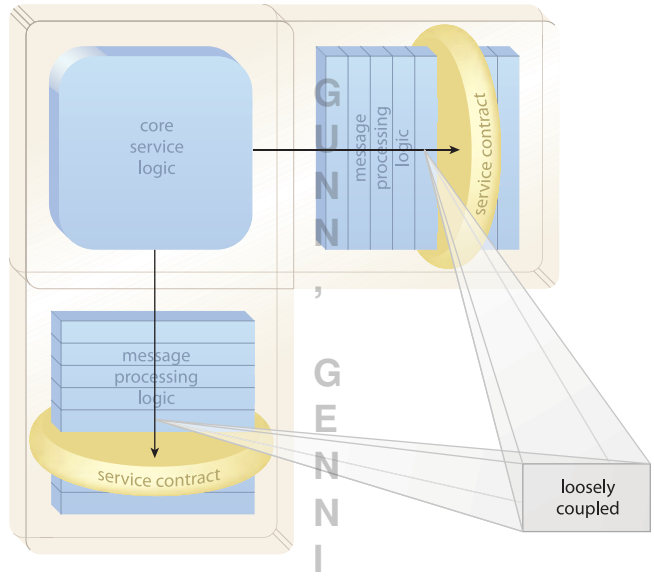

Figure 7.22
The same core service logic now accessible via two separate service contracts. This effectively establishes two Web services based on the same underlying logic.

Note also that this form of logic normalization (using the same underlying service logic in support of multiple services) can result from the discovery of similar but physically separate services and the application of refactoring-related design patterns to consolidate service logic into one location.

Problems when Schema Coupling Is "too loose"

Sometimes in pursuit of reducing consumer dependencies, contract schemas are "over-streamlined," resulting in a weakly typed, bare bones data model that does little more than establish some very generic data types.

The rationale behind this approach is to enable the service to accept and transmit a range of data via request and response messages, allowing both service and consumer owners to make more changes without affecting the published service contract.

By building in too much flexibility, the service logic is required to perform extra runtime processing just to interpret the data it receives for any given invocation instance. The net result of over-emphasizing a lower consumer coupling level can therefore increase service performance requirements.

Furthermore, the less that is published in a service contract, the more consumer programs may need to know about how the underlying solution logic is designed. Depending on the nature of the logic, this can lead to undesirable forms of implementation coupling.

SUMMARY OF KEY POINTS

- A common design concern with the logic-to-contract approach is limiting the logic to just one service contract.
- Another possible risk associated with loosely coupled consumer relationships is the introduction of runtime performance overhead and inadvertent implementation coupling.

7.7 CASE STUDY EXAMPLE

The following three services defined by Cutit Saws in the case study example from Chapter 6 are revisited to ensure that appropriate coupling levels are implemented:

- Materials Service
- Formulas Service
- Run Lab Project Service

Coupling Levels of Existing Services

Because all three are custom services for which standardized service contracts were delivered, each exhibits a high level of logic-to-contract coupling and negligible contract-to-logic coupling.

The Materials and Formulas services were based on the entity service model, which deliberately decreases potential functional coupling to external or parent business process logic.