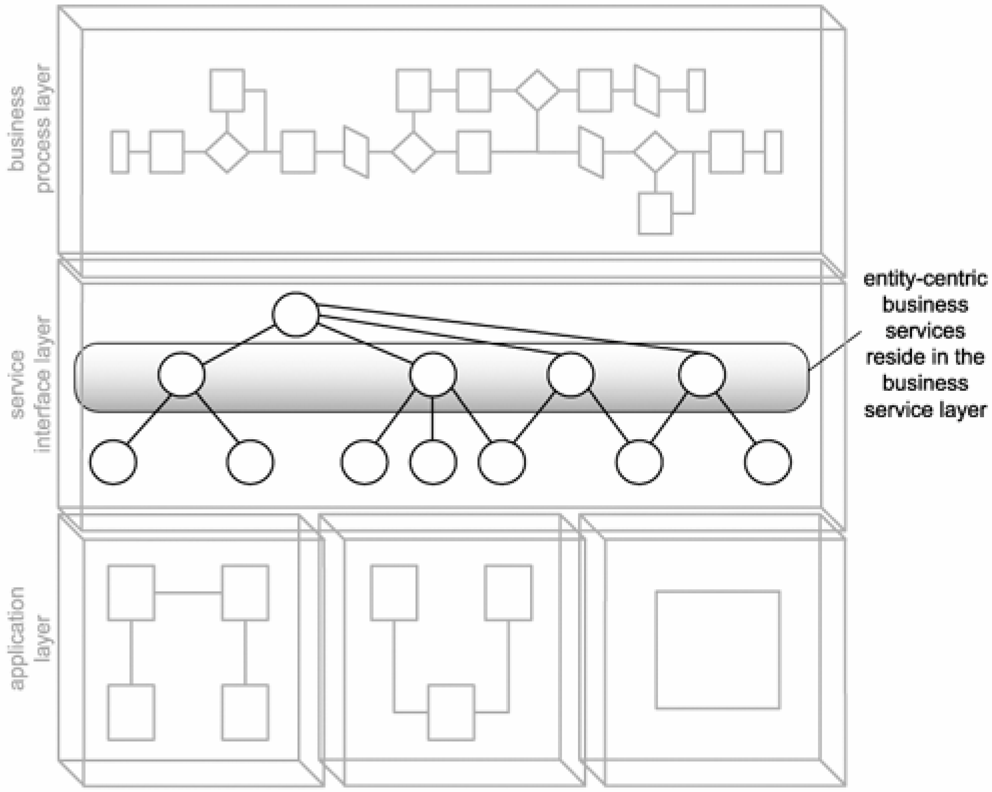
Entity-centric business services

Entity-centric business services represent the one service layer that is the least influenced by others. Its purpose is to accurately represent corresponding data entities defined within an organization's business models. These services are strictly solution- and business process-agnostic, built for reuse by any application that needs to access or manage information associated with a particular entity.

Figure 15.1. Entity-centric services establish the business service layer.

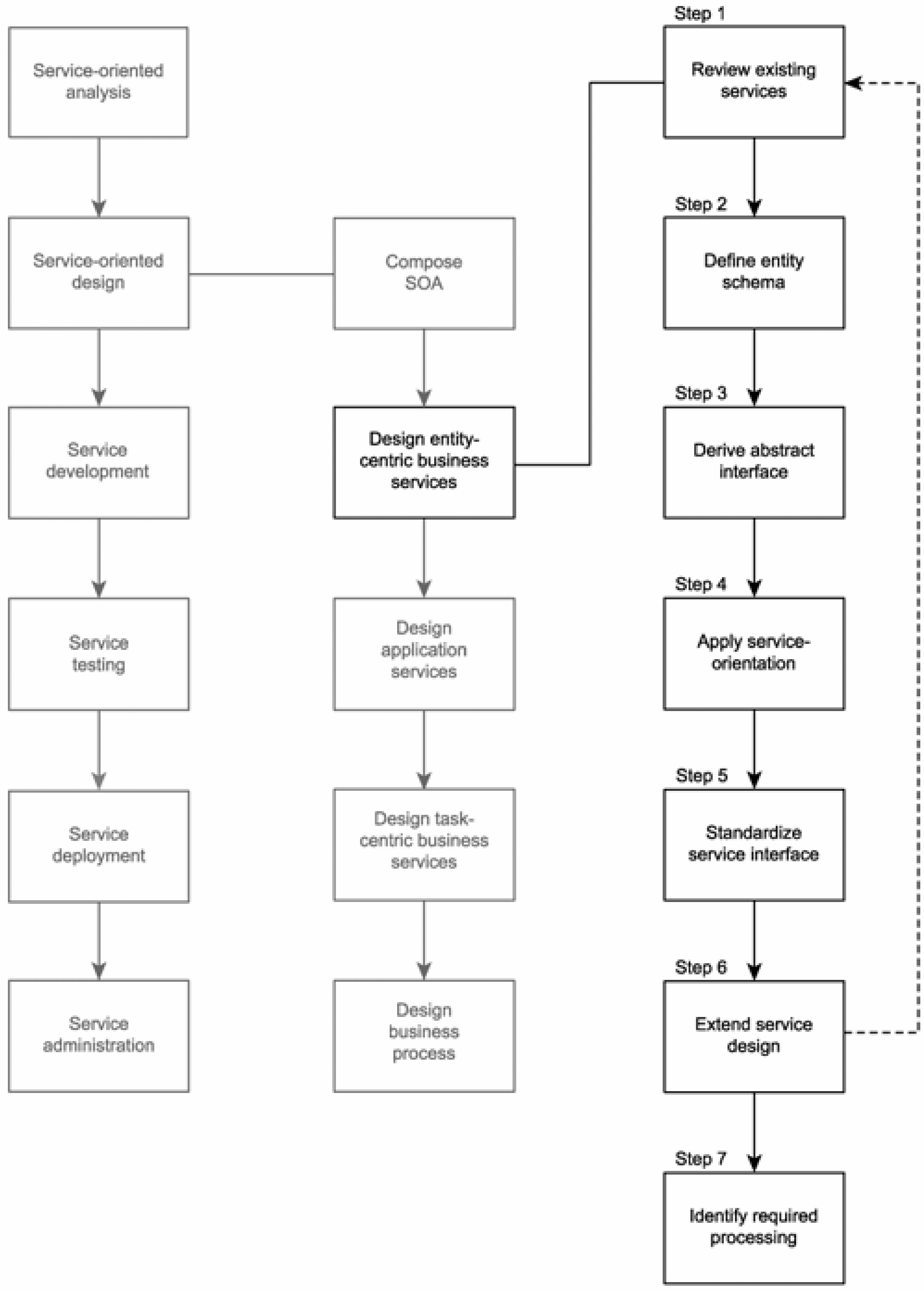

business process layer
service interface layer
application layer
entity-centric business services reside in the business service layer

Because they exist rather atomically in relation to other service layers, it is beneficial to design entity-centric business services prior to others. This establishes an abstract service layer around which process and underlying application logic can be positioned.

15.2.1. Process description

Provided next is the step-by-step process description wherein we establish a recommended sequence of detailed steps for arriving at a quality entity-centric business service interface (Figure 15.2).

Figure 15.2. The entity-centric business service design process.



Note that the order in which these steps are provided is not set in stone. For example, you may prefer to define a preliminary service interface prior to establishing the actual data types used to represent message body content. Or perhaps you may find it more effective to perform a speculative analysis to identify possible extensions to the service before creating the first cut of the interface.

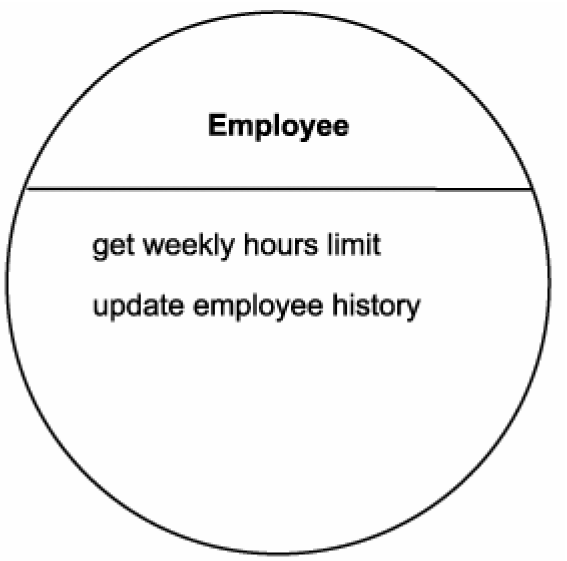
All of these can be legitimate approaches. The key is to ensure that in the end, design standards are applied equally to all service operations and that all processing requirements are accurately identified.

Let's begin now with the design of our entity-centric business service.

Case Study

The examples provided alongside this process description revisit the TLS environment. Specifically, we take another look at the Employee Service candidate that was modeled at the end of Chapter 12 (Figure 15.3).

Figure 15.3. The Employee Service candidate.



The Employee Service was modeled intentionally to facilitate an entity-centric grouping of operations. As part of the Timesheet Submission Process, this service is required to contribute two specific functions.

The first requires it to execute a query against the employee record to retrieve the maximum number of hours the employee is authorized to work within a week. The other piece of functionality it needs to provide is the ability to post updates to the employee's history. As you may recall from the original Timesheet Submission Process, this action is required only when a timesheet is rejected.

The result of the TLS service modeling process was to express these two functions through the assignment of the following two operation candidates:

- get weekly hours limit
- update employee history

This service candidate now provides us with a primary input from which we derive a service design by following the steps in the following entity-centric business service design process.

Note

The "get weekly hours limit" operation candidate (which later becomes the getWeeklyHoursLimit operation) proposes an unusually fine-grained operation. Service operations, in general, tend to be more coarse-grained to overcome the performance overhead associated with SOAP message exchanges. For simplicity's sake, we retain the granularity of this operation, as it fulfills the functional requirements of our business process. For more information regarding service interface granularity, see the Apply a suitable level of interface granularity guideline at the end of this chapter.

Step 1: Review existing services

Ideally, when creating entity-centric services, the modeling effort resulting in the service candidates will have taken any existing services into account. However, because service candidates tend to consist of operation candidates relevant to the business requirements that formed the basis of the service-oriented analysis, it is always worth verifying to ensure that some or all of the processing functionality represented by operation candidates does not already exist in other services.

Therefore, the first step in designing a new service is to confirm whether it is actually even necessary. If other services exist, they may already be providing some or all of the functionality identified in the operation candidatesorthey may have already established a suitable context in which these new operation candidates can be implemented (as new operations to the existing service).

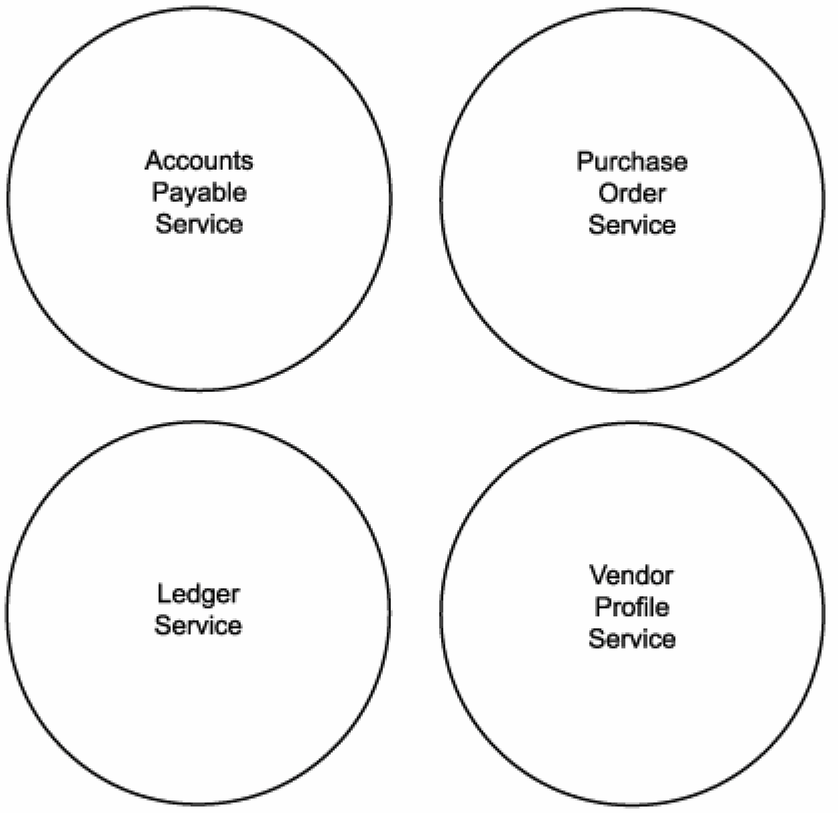
Case Study

This is only TLS's second service-oriented solution. The first was created to establish an external interface with their accounting system via a B2B environment. That solution was built according to the top-down delivery strategy and therefore resulted in a collection of entity-centric business services.

Architects involved with the service design for this new Timesheet Submission application are pretty confident that the new set of services in no way overlaps with the existing ones. However, because the B2B solution was built by a completely different project team, they agree that it is worth the effort to review existing services before commencing with the design process.

Their investigation reveals that the following entity-centric business services were delivered as part of the B2B project: Accounts Payable Service, Purchase Order Service, Ledger Service, and Vendor Profile Service (Figure 15.4).

Figure 15.4. The existing inventory of TLS services.



It appears evident by the naming alone that each service represents an entity separate and distinct from the Employee entity proposed by the current service candidate. Just to be sure, though, each service description (along with any supplemental metadata) is reviewed. The project architects then conclude that no overlap exists, which gives them the green light to proceed with the design of the Employee Service.

Step 2: Define the message schema types

It is useful to begin a service interface design with a formal definition of the messages the service is required to process. To accomplish this we need to formalize the message structures that are defined within the WSDL `types` area.

SOAP messages carry payload data within the `Body` section of the SOAP envelope. This data needs to be organized and typed. For this we rely on XSD schemas. A standalone schema actually can be embedded in the `types` construct, wherein we can define each of the elements used to represent data within the SOAP body.

In the case of an entity-centric service, it is especially beneficial if the XSD schema used accurately represents the information associated with this service's entity. This "entity-centric schema" can become the basis for the service WSDL definition, as most service operations will be expected to receive or transmit the documents defined by this schema.

Note that there is not necessarily a one-to-one relationship between entity-centric services and the entities that comprise an entity model. You might recall in the service modeling example from Chapter 12, we combined Employee and EmployeeHistory entities into one Employee Service. In this case, you can either create two separate schemas or combine them into one. The latter option is recommended only if you are confident you will never want to split these entities up again.

Note

As demonstrated in the upcoming example, the WSDL definition can import schemas into the `types` area. This can be especially beneficial when working with standardized schemas that represent entities. (See the Consider using modular WSDL documents guideline for more information.)

Case Study

TLS invested in creating a standardized XML data representation architecture (for their accounting environment only) some time ago. As a result, an inventory of entity-centric XSD schemas representing accounting-related information sets already exists.

At first, this appears to make this step rather simple. However, upon closer study, it is discovered that the existing XSD schema is very large and complex. After some discussion, TLS architects decidefor better or for worsethat they will not use the existing schema with this service at this point. Instead, they opt to derive a lightweight (but still fully compliant) version of the schema to accommodate the simple processing requirements of the Employee Service.

They begin by identifying the kinds of data that will need to be exchanged to fulfill the processing requirements of the "Get weekly hours limit" operation candidate. They end up defining two complex types: one containing the search criteria required for the request message received by the Employee Service and another containing the query results returned by the service. The types are deliberately named so that they are associated with the respective messages. These two types then constitute the new Employee.xsd schema file.

Example 15.1. The Employee schema providing `complexType` constructs used to establish the data representation anticipated for the "Get weekly hours limit" operation candidate.

Note

The `complexType` constructs are wrapped in `element` constructs to comply with WS-I requirements for document + literal SOAP messages.

However, just as the architects attempt to derive the types required for the "Update employee history" operation candidate, another problem presents itself. They discover that the schema from which they derived the Employee.xsd file does not represent the

EmployeeHistory entity, which this service candidate also encapsulates.

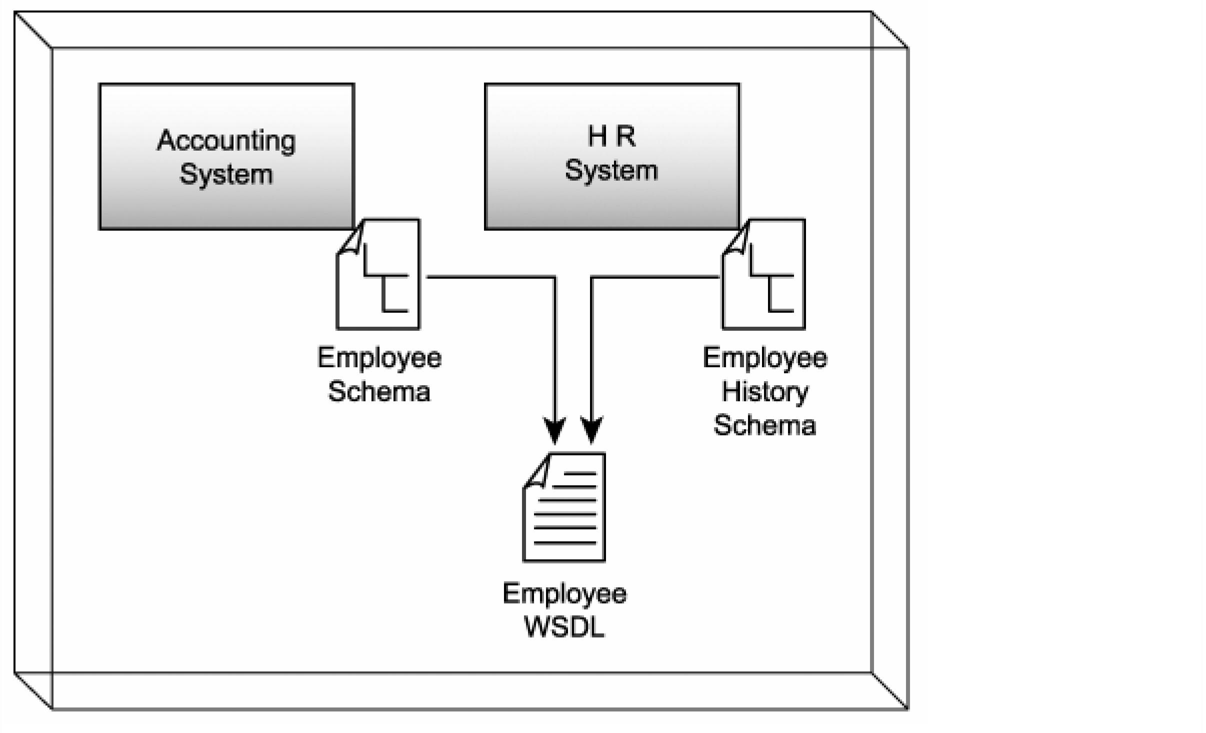
Another visit to the accounting schema archive reveals that employee history information is not governed by the accounting solution. It is, instead, part of the HR environmentfor which no schemas have been created.

Not wanting to impose on the already standardized design of the Employee schema, it is decided that a second schema definition be created (Figure 15.5). Analysts get involved and produce the following EmployeeHistory.xsd schema:

Example 15.2. The EmployeeHistory schema, with a different `targetNamespace` to identify its distinct origin.

```
targetNamespace=
 "http://www.xmltc.com/tls/employee/schema/hr/">
```

Figure 15.5. Two schemas originating from two different data sources.



To promote reusability and to allow for each schema file to be maintained separately from the WSDL, the XSD schema `import` statement is used to pull the contents of both schemas into the Employee Service WSDL `types` construct.

Example 15.3. The WSDL `types` construct being populated by imported schemas.

```
import namespace=
"http://www.xmltc.com/tls/employee/schema/accounting/"
schemaLocation="Employee.xsd"/>
import namespace=
"http://www.xmltc.com/tls/employee/schema/hr/"
schemaLocation="EmployeeHistory.xsd"/>
```

Step 3: Derive an abstract service interface

Next, we analyze the proposed service operation candidate and follow these steps to define an initial service interface:

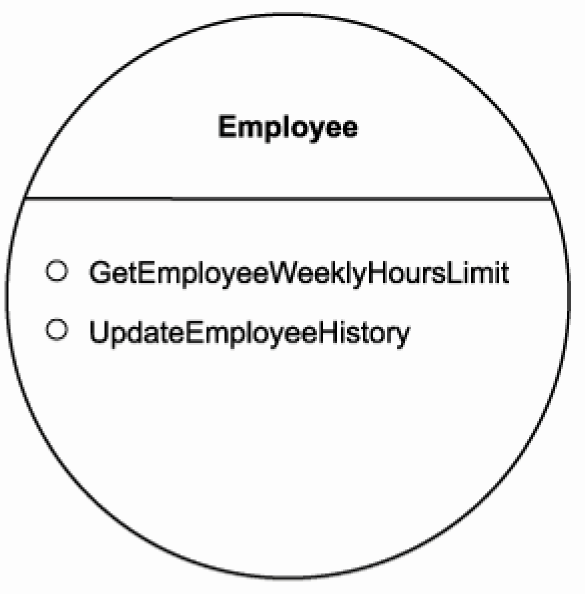
1. Confirm that each operation candidate is suitably generic and reusable by ensuring that the granularity of the logic encapsulated is appropriate. Study the data structures defined in Step 2 and establish a set of operation names.
2. Create the `portType` (or `interface`) area within the WSDL document and populate it with `operation` constructs that correspond to operation candidates.

3. Formalize the list of input and output values required to accommodate the processing of each operation's logic. This is accomplished by defining the appropriate `message` constructs that reference the XSD schema types within the child `part` elements.

Case Study

The TLS architects decide on the following operations names: GetEmployeeWeeklyHoursLimit and UpdateEmployeeHistory (Figure 15.6).

Figure 15.6. The Employee Service operations.



They subsequently proceed to define the remaining parts of the abstract definition, namely the `message`, and `portType` constructs.

Example 15.4. The `message` and `portType` parts of the Employee Service definition that implement the abstract definition details of the two service operations.

Note

TLS has standardized on the WSDL 1.1 specification because it is conforming to the requirements dictated by version 1.1 of the WS-I Basic Profile and because none of its application platforms support a newer WSDL version. WSDL 1.1 uses the `portType` element instead of the `interface` element, which is introduced by WSDL 2.0.

Step 4: Apply principles of service-orientation

Here's where we revisit the four service-orientation principles we identified in Chapter 8 as being those not provided by the Web services technology set:

- service reusability
- service autonomy
- service statelessness
- service discoverability

Reusability and autonomy, the two principles we already covered in the service modeling process, are somewhat naturally part of the entity-centric design model in that the operations exposed by entity-centric business services are intended to be inherently generic and reusable (and because the use of the `import` statement is encouraged to reuse schemas and create modular WSDL definitions). Reusability is further promoted in Step 6, where we suggest that the design be extended to facilitate requirements beyond those identified as part of our service candidate.

Because entity-centric services often need to be composed by a parent service layer and because they rely on the application service layer to carry out their business logic, their immediate autonomy is generally well defined. Unless those services governed by an entity-centric controller have unusual processing requirements or impose dependencies in some manner, entity-centric services generally maintain their autonomy.

It is for similar reasons as those just mentioned that statelessness is also relatively manageable. Entity-centric services generally do not possess a great deal of workflow logic and for those cases in which multiple application or business services need to be invoked to carry out an operation, it is preferred that state management be deferred as much as possible (to, for example, document-style SOAP messages).

Discoverability is an important part of both the design of entity-centric services and their post-deployment utilization. As we mentioned in Step 1, we need to ensure that a service design does not implement logic already in existence. A discovery mechanism would make this determination much easier. Similarly, one measure we can take to make a service more discoverable to others is to supplement it with metadata details using the `documentation` element, as explained in the Document services with metadata guideline.

Case Study

Upon a review of the initial abstract service interface, it is determined that a minor revision can be incorporated to better support fundamental service-orientation. Specifically, meta information is added to the WSDL definition to better describe the purpose and function of each of the two operations and their associated messages.

Example 15.5. The service interface, supplemented with additional metadata documentation.

```
<documentation>
GetEmployeeWeeklyHoursLimit uses the Employee
ID value to retrieve the WeeklyHoursLimit value.
```

```
UpdateEmployeeHistory uses the Employee ID value
to update the Comment value of the EmployeeHistory.
documentation>
```

Step 5: Standardize and refine the service interface

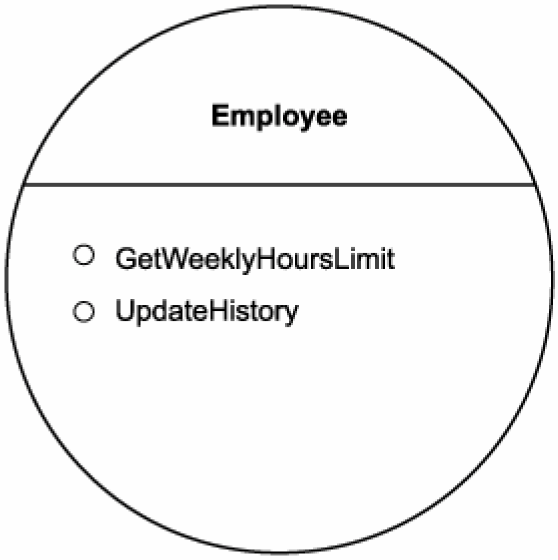
Depending on your requirements, this can be a multi-faceted step involving a series of design tasks. Following is a list of recommended actions you can take to achieve a standardized and streamlined service design:

- Review existing design standards and guidelines and apply any that are appropriate. (Use the guidelines and proposed standards provided at the end of this chapter as a starting point.)
- In addition to achieving a standardized service interface design, this step also provides an opportunity for the service design to be revised in support of some of the contemporary SOA characteristics we identified in the Unsupported SOA characteristics section of Chapter 9.
- If your design requirements include WS-I Basic Profile conformance, then that can become a consideration at this stage. Although Basic Profile compliance requires that the entire WSDL be completed, what has been created so far can be verified.

Case Study

The TLS architect in charge of the Employee Service design decides to make adjustments to the abstract service interface to apply current design standards. Specifically, naming conventions are incorporated to standardize operation names, as shown in Figure 15.7.

Figure 15.7. The revised Employee Service operation names.



Example 15.6. The two `operation` constructs with new, standardized names.

As explained in the Apply naming standards guideline, the use of naming standards provides native support for intrinsic interoperability, a key contemporary SOA characteristic.

Step 6: Extend the service design

The service modeling process tends to focus on evident business requirements. While promoting reuse always is encouraged, it often falls to the design process to ensure that a sufficient amount of reusable functionality will be built into each service. This is especially important for entity-centric business services, as a complete range of common operations typically is expected by their service requestors.

This step involves performing a speculative analysis as to what other types of features this service, within its predefined functional context, should offer.

There are two common ways to implement new functionality:

- add new operations
- add new parameters to existing operations

While the latter option may streamline service interfaces, it also can be counter-intuitive in that too many parameters associated with one operation may require that service requestors need to know too much about the service to effectively utilize it.

Adding operations is a straight-forward means of providing evident functions associated with the entity. The classic set of operations for an entity-centric service is:

- GetSomething
- UpdateSomething
- AddSomething
- DeleteSomething

Security requirements notwithstanding, establishing these standard operations builds a consistent level of interoperability into the business service layer, facilitating ad-hoc reusability and composition.

Note

Despite the naming suggestions listed here, when designing business services to reflect existing entity models, it is often beneficial to carry over the naming conventions already established (even if this means adjusting existing naming standards accordingly).

If entirely new tasks are defined, then they can be incorporated by new operations that follow the same design standards as the existing ones. If new functional requirements are identified that relate to existing operations, then a common method of extending these operations is to add input and output values. This allows an operation to receive and transmit a range of message combinations. Care must be taken, though, to not overly complicate operations for the sake of potential reusability. It often is advisable to subject any new proposed functionality to a separate analysis process.

Also, while it is desirable and recommended to produce entity-centric services that are completely self-sufficient at managing data associated with the corresponding entity domain, there is a key practical consideration that should be factored in. For every new operation you add, the means by which that operation completes its processing also needs to be designed and implemented. This boils down to the very probable requirement for additional or extended application services. As long as the overhead for every new operation is calculated and deemed acceptable, then this step is advisable.

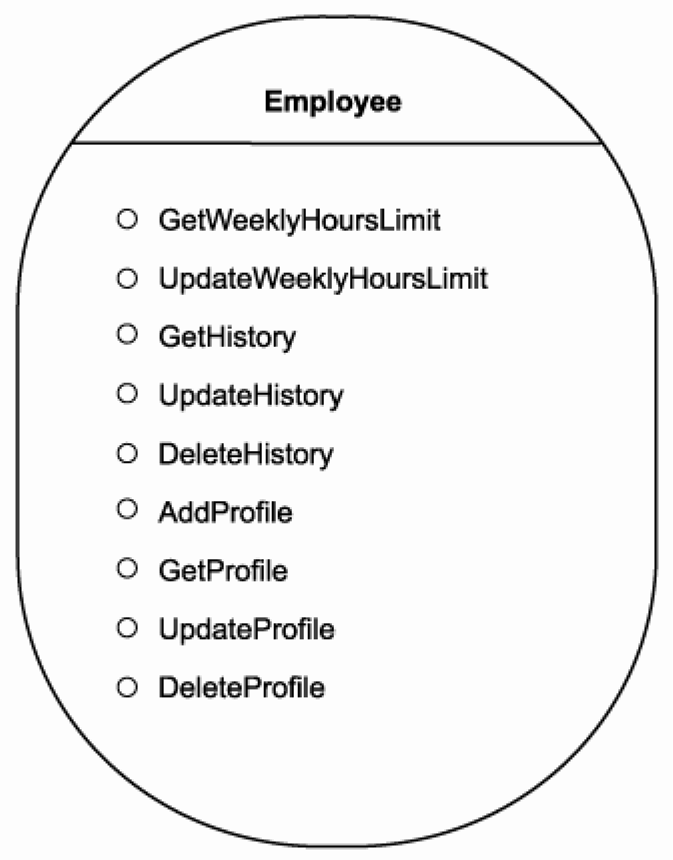
Note that upon identifying new operations, Steps 1 through 5 need to be repeated to properly shape and standardize added extensions.

Case Study

TLS is under time pressure to deliver the Timesheet Submission solution. It is therefore decided that they will not extend the service design at this point. The standards applied so far have guaranteed them an easily extensible service design, where additional operations can be added without breaking the original service interface.

However, to demonstrate this step, let's speculate as to the kinds of operations that could have been added to the Employee Service. Given that this service represents two entities, it likely will require a larger range of operations than most entity-centric services, as shown in Figure 15.8.

Figure 15.8. An Employee Service offering a full range of operations.



Following is an example that shows how the `portType` construct could be expanded with supplementary operations (`documentation` elements have been omitted to save space):

Example 15.7. An expanded `portType` construct.

These additional operations provide a well rounded set of data processing extensions that enable the Employee Service to be reused by a variety of solutions.

Step 7: Identify required processing

While the service modeling process from our service-oriented analysis may have identified some key application services, it may not have been possible to define them all.

Now that we have an actual design for this new business service, you can study the processing requirements of each of its operations more closely. In doing so, you should be able to determine if additional application services are required to carry out each piece of exposed functionality. If you do find the need for new application services, you will have to determine if they already exist, or if they need to be added to the list of services that will be delivered as part of this solution.

Case Study

Let's take another look at the two operations we designed into the Employee Service:

- GetWeeklyHoursLimit
- UpdateHistory

The first requires that we access the employee profile. At TLS, employee information is stored in two locations:

- Payroll data is kept within the accounting system repository, along with additional employee contact information.
- Employee profile information, including employee history details, is stored in the HR repository.

When an XML data representation architecture was first implemented at TLS, entity-centric XSD schemas were used to bridge some of the existing disparity that existed among the many TLS data sources. Being aware of this, the service architect investigates the origins of the Employee.xsd schema used as part of the Employee.wsdl definition to determine the processing requirements for the GetWeeklyHoursLimit operation.

It is discovered that although the schema accurately expresses a logical data entity, it represents a document structure derived from two different physical repositories. Subsequent analysis reveals that the weekly hours limit value is stored in the accounting database. The processing requirement for the GetWeeklyHoursLimit operation is then written up as follows:

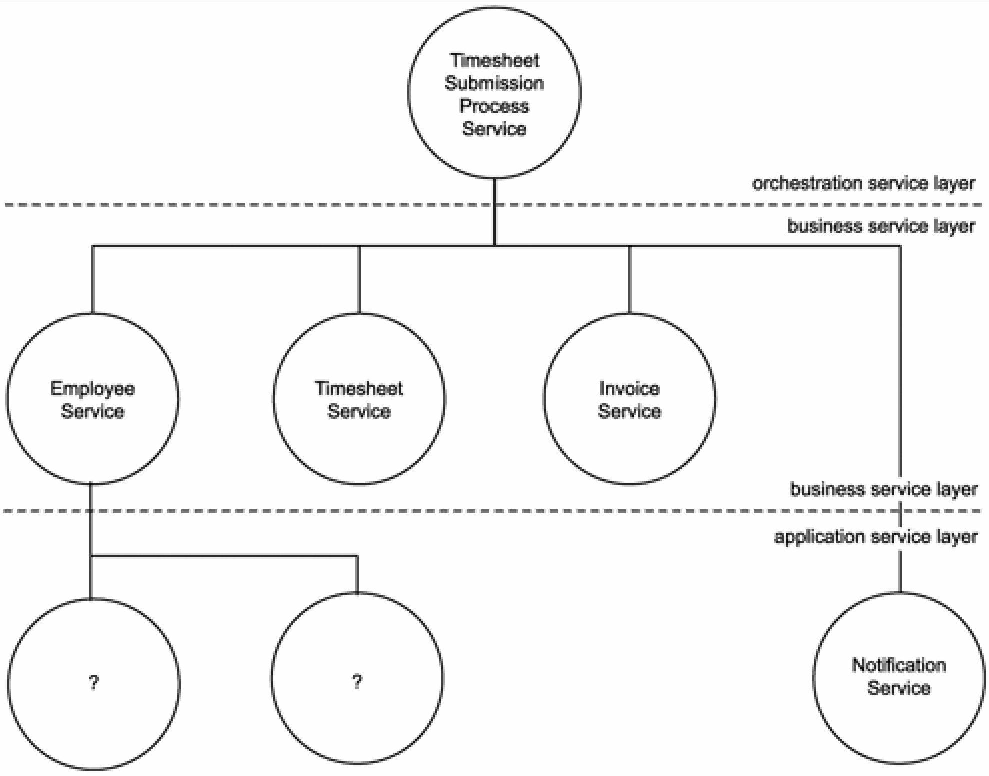
Application service-level function capable of issuing the following query against the accounting database: "Return employee's weekly hour limit using the employee ID as the only search criteria."

Next, the details behind the UpdateHistory operation are studied. This time it's a bit easier, as the EmployeeHistory.xsd schema is associated with a single data sourcethe HR employee profile repository. Looking back at the original analysis documentation, the architect identifies the one piece of information that this particular solution will need to update within this repository. Therefore, the processing requirement definition goes beyond the immediate requirements of the solution, as follows:

Application service-level function capable of issuing an update to the "comment" column of the employee history table in the HR employee profile database, using the employee ID value as the sole criteria.

It looks like the Timesheet Submission solution may require new application services to facilitate Employee Service processing requirements, as illustrated in the expanded composition shown in Figure 15.9. Both of these newly identified requirements will need to be subjected to the service modeling process described in Chapter 12.

Figure 15.9. The revised composition hierarchy identifying new potential application services.



It is eventually revealed that only one new application service is required to accommodate the Employee Servicea Human Resources wrapper service that also can facilitate the Timesheet Service. Additionally, it is discovered that the processing required by the Invoice Service can be fulfilled by the existing TLS Accounts Payable Service.

Case Study (Process Results)

Here is the final version of the Employee Service definition, incorporating the changes to element names and all of the previous revisions.

Example 15.8. The final abstract service definition.

```
GetWeeklyHoursLimit uses the Employee ID value
to retrieve the WeeklyHoursLimit value.
UpdateHistory uses the Employee ID value to
update the Comment value of the EmployeeHistory.
```

```
...
```

Note

This process has produced an abstract definition only. The full WSDL document, including concrete definition details, along with the imported XSD schemas, can be downloaded at `www.serviceoriented.ws`.

SUMMARY OF KEY POINTS

- Entity-centric business services need to be designed to accurately represent existing business entities, while remaining business process-agnostic.
- Some speculative analysis may be required to properly outfit an entity-centric business service with the required range of generic operations.