Capturing and analyzing interface characteristics, Part 2: Reference guide to integration characteristics

Kim J. Clark  
Brian M. Petrini  
January 25, 2012

Part 1 of this two-part article discussed the definition of integration characteristics and how they are best used to reduce risk and improve the efficiency of design for the integration aspects of a solution. Part 2 provides detailed reference information about the integration characteristics themselves to ensure a clear and common understanding of the meaning, importance, and use of each characteristic.

View more content in this series

Introduction

Understanding interface characteristics is fundamental to understanding how systems interact with one another. Whilst some integration specifics, such as protocols, data formats, reference architectures, and methodologies, come and go over time, the characteristics of integration between systems have remained largely the same.

The full set of interface characteristics is shown below:

A. FUNCTIONAL DEFINITION
   • Principal data objects
   • Operation/function
   • Read or change
   • Request/response objects

B. TECHNICAL INTERFACE
   • Transport
   • Protocol
   • Data format

C. INTERACTION TYPE
   • Request-response or fire-forget
   • Thread-blocking or asynchronous
   • Batch or individual
   • Message size
D. PERFORMANCE
- Response times
- Throughput
- Volumes
- Concurrency

E. INTEGRITY
- Validation
- Transactionality
- Statefulness
- Event sequence
- Idempotence

F. SECURITY
- Identity/authentication
- Authorization
- Data ownership
- Privacy

G. RELIABILITY
- Availability
- Delivery assurance

H. ERROR HANDLING
- Error management capabilities
- Known exception conditions
- Unexpected error presentation

This article is essentially a reference guide to these integration characteristics, which were introduced in Part 1. For each characteristic, this article provides:

- **Description**: A brief explanation or question to describe what the characteristic refers to.
- **Example**: A real world example of what you might capture for this characteristic.
- **You need to know**: Additional information that either quantifies the characteristic in more depth, or alludes to some of the subtleties involved in capturing it.
- **Why do you care?**: A brief description of why this characteristic is important, and what the consequences might be if you do not capture it.

Individually, each interface characteristic listed in the table above certainly seems essential. There are probably even more characteristics that could be captured, but the importance of those shown here have been demonstrated time again. All have been shown to have a significant impact on integration design, and a critical impact on the estimation of complexity and risk. These characteristics have been captured through years of experience from consultants covering vastly different technical solutions in a variety of industries. It is worth bearing in mind that every one of these characteristics is responsible for a project emergency somewhere that occurred as a result of not capturing it with sufficient clarity at the right time.

The interface characteristics are presented here in subjective groupings to make it easier to explain, remember, and interpret this information.
A. Functional definition

Principal data objects

**Description:** The names of the primary data objects (sometimes referred to as entities) that can be exchanged via the available operations on this interface to the back end system.

**Example:** A back end system can predominantly manage data objects such as Customer, Order, Invoice, or Product, and, as such, these are the principle data objects available via its interfaces, as shown in Figure 1.

**You need to know:** These are logical objects so you can generally compare what is stored where. You also want to establish whether ownership of these entities is clear; for example, is the system exposing the interface formally the master for this data, or is it a replica? For this reason, the names of the entities should be ideally taken from a global business vocabulary (or object model) used by the enterprise to improve consistency.

**Why do you care?** There will be a constant need throughout your project to have good conceptual understanding of where you find entities from the global data model within the various systems. If entities are present in more than one system, you can quickly begin the discussion around who owns the "master" of a given piece of data for the enterprise, and whether there is a data synchronisation strategy for this entity.

Operation/function

**Description:** Ultimately, you need to know the specific actions that can be performed on or by each of the back end systems to fulfill each step in a business process. This becomes the list of "actions" that you have at your disposal, as shown in Figure 2.
Figure 2. Examples of operation granularity

Example: Operations required to fulfill a business process might be: submitApplication, getQuote, createCustomer, sendConfirmationLetter. Notice that the operation names are typically expressed as close to a simple verb + noun pair as possible.

You need to know: In a "perfect" SOA, you would simply be able to find all these operations from within a single service registry. In reality, only some will be available from the registry. For others, you will need to investigate the back end system's interface documentation (if present). You will often find that the operations on back end systems are at a different level of granularity than what you require, as shown in Figure 2 — or worse, they might not yet exist at all.

Why do you care? This is one of the most important checks for completeness. If the operations are not available, you will have to create them. Sometimes, the data might not even be available at all. Equally important, drilling into the specific operations will show you whether the granularity of the actions available on the back end system interfaces is the same as that required by the business processes in which they are involved. If the granularity is different, this is an early warning that composition or orchestration is required, adding to the complexity of the work.

Read or change
Description: Does the operation result in a change of state of the underlying data? Is the data stored within (or manipulated by) the back end system changed as a result of the operation? If so, it is a change. If not, it is a read.

Example: From the traditional logical operations known as CRUD (create/read/update/delete), a change refers to a create, update, or delete. If you purely retrieve data with no change of state, then it is a read.

You need to know: This characteristic can often be inferred from the operation or function name or description, but not with certainty. It is critical to evaluate each operation independently. For example, it might seem inevitable that the operation getAddressForContact was a read. Consider, however, if the system was for a police force and the type of contact was police informers. It is very likely that the system would want to audit every read for security reasons so that you know who is...
asking for addresses of the informers. The recording of that audit trail means that the operation in this context is, in fact, a change.

**Why do you care?** This characteristic has important knock-on effects on other characteristics, and it has a huge effect on the integration patterns used. Read operations can appear simpler to design, because, for example, they do not usually require transactions (though there are special cases, as noted in the Transactionality characteristic); they are implicitly stateless, less likely to require auditing, and often available to a lower security level. However, they can be equally or more complex; for example, if the operations are always request-response requiring correlation in some way. They can have shorter response time requirements (as they are often used by graphical user interfaces), have typically higher throughput rate, and are more regularly used, forcing the need for performance enhancements such as caching.

**Request/response objects**

**Description:** When you use this operation, what data do you need to send, and what will you get in return?

**Example:** Initially, you can simply embellish the operation name to create a full method signature, as shown in Figure 2. For example, getLoanQuote(LoanDetails, Customer) returns LoanQuote. Ultimately, however, you will need to fully specify the request and response objects down to the level of their attributes, plus those on any child objects, as shown in Tables 1 and 2.

**Table 1. Example of simple specification of request objects for the operation createCustomer**

<table>
<thead>
<tr>
<th>REQUEST</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer.firstName</td>
<td>Mandatory. Customers first name. &lt; 32 characters.</td>
</tr>
<tr>
<td>Customer.secondName</td>
<td>Mandatory. Customers second name. &lt; 32 characters.</td>
</tr>
<tr>
<td>Customer.dateOfBirth</td>
<td>Mandatory. Customers date or birth in the form mm/dd/yyyy.</td>
</tr>
<tr>
<td>Customer.address.addressLine1</td>
<td>Mandatory. First line of customers address. &lt; 32 characters.</td>
</tr>
<tr>
<td>Customer.address.addressLine2</td>
<td>Optional. Second line of customers address. &lt; 32 characters.</td>
</tr>
<tr>
<td>and so on.</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2. Example of simple specification of response object for the operation createCustomer**

<table>
<thead>
<tr>
<th>RESPONSE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer.referenceNumber</td>
<td>Always returned. Unique reference to the customer in the form Cnnnnnnn, where nnnnnnn is a unique number.</td>
</tr>
</tbody>
</table>

**You need to know:** If these objects are taken from a generalised (or global) object model containing all possible attributes, then you will also need to know which attributes are used in this specific operation and whether they are mandatory. This can be difficult to represent. XML schema, for example, which is commonly used to specify data models in SOA, enables you to specify whether attributes are mandatory or optional, but only once for a given object. At the time
of this writing, XML schema does not provide a way of specifying restrictions to the use of an object for a specific context.

Multiple cardinality objects (arrays/lists) need special attention. For example, what does a list mean in a data changing operation? Consider where the provided list overlaps incompletely with the existing list. Do you then need to translate that into a list merge involving a combination of creates, updates, and deletes?

Finally, you need to consider which unique references and enumerated values are used in the objects, as these might not be generally known or available outside the provider.

**Why do you care?** If you do not go to this level of depth, you cannot be sure that all the necessary data is available at each step in the process. Without detailed knowledge of the interface request/response objects, you will be unable to design the mapping required from the requester’s data model to the interface data model. The importance of this is often understated, and many projects stall at the detailed design because those who truly understand the data are no longer available to the project.

Also, knowing what is sent to and received from a back end system when invoking a particular operation is critical for validation of the business process flow. How else can you be sure that you have all the data you need at each stage to perform the next action in line? Initially, you can consider this at entity level, but very quickly it becomes important to look down to the individual attribute level. For example, if a key piece of data (which could be a single attribute on a complex object such as "partner's product code" on an "order line" object within an "order") is not present, then you might not be able to make the next invocation to "processOrderFromPartner." Without that single piece of data, the process flow is completely invalid and might require an extra system call — or worse, human intervention — before the process can actually work.

**B. Technical interface**

**Transport**

**Description:** The transport that requesters use to connect to the system for this interface. A transport is the medium used by the interface to transfer data from requestor to provider and back. You could ask the question, "What carries the data between requestor and provider?"

**Example:** HTTP, IIOP, WebSphere MQ, TCP/IP.

**You need to know:** There are two primary transport types, as shown below and in Figure 3. It will be important to understand these differences for the discussion later about whether the overall interaction type is **thread blocking** or **synchronous**.
Figure 3. The two most common transport types

1) **Non-blocking (messaging) transport**: A messaging transport, such as IBM® WebSphere® MQ, enables the requester to simply place the message on a nearby message queue. This is considered **non-blocking** because the requester need only wait for the message to be placed on the queue and then it is free to do other work; in other words, it is not “blocked.” The provider might not even be available at this point, and the requester certainly has no concept of where the provider is located. The provider can process the messages at whatever time, and in whatever order it chooses. There are always multiple transactions involved in a messaging-based interaction. Using a messaging-based transport is part of what is required to implement many of the more advanced integration patterns. Messaging provides excellent decoupling of requester and provider.

2) **Blocking (synchronous) transport**: A synchronous transport, such as HTTP, holds a channel open between requester and provider whilst the invocation is being processed. The requester awaits a response from the provider containing data, or at least an acknowledgement that the request has been fulfilled. The provider must process the invocation immediately. It is possible to complete a blocking request in a single transaction if both requester and provider can participate in global transactions, and the transport has a transactional protocol. Synchronous transport provides a relatively tight coupling between requester and provider.

You can easily get tied up trying too hard to separate transport and protocol. For example, a complication here is that something that is a transport for one interface could be considered the protocol for another. For SOAP/HTTP, SOAP is the protocol, and HTTP is then just seen as the transport. However, in a REST interaction, HTTP could be seen as the protocol itself, with TCP/IP as the transport. Going deeper still, for an interface specified purely over TCP/IP, TCP is in fact the protocol, and IP the transport. Confused? Most people are. Ultimately, don’t try to be too precise here; just capture a meaningful term about the protocol and transport that will make sense to most users of the characteristics. Typically, the top two capabilities required to make connectivity over the interface are the relevant protocol and transport, respectively (for example, SOAP/HTTP, or RMI/IIOP, or XML/MQ).

**Why do you care?** Differences in transport are the first hurdle you must cross in integration. You cannot even make two systems connect unless you can find a common transport. It is, therefore, one of the first characteristics that we standardise on to make SOA style services more reusable. As a result of the ubiquity of the web, HTTP transports, for example, are readily available to most modern requesters. If the transport offered by a provider is obscure, it is likely you will require a transport switch integration pattern. You need to know if your middleware can be easily switched between these protocols or whether you are in unproven territory.
Protocol

Description: A protocol is essentially an agreement between requestor and provider about how and when they will send each other data over the transport, and how the data should be interpreted at the other end. What (if any) protocol is used on top of the transport to control the message conversation?

Example: Some interactions might have no protocol at all. A simple one way interaction using XML over a WebSphere MQ transport has no formal protocol. However, it is also very limited. Web services, however, require a more formalized interaction style to make them more extensible and re-usable. The protocol commonly used for web service is SOAP. SOAP provides a structure to request and response messages so that they can be seen as part of the same interaction, and that they can carry additional data, such as attachments. The protocol also defines standard headers that can be used to instruct how the message should be interpreted, and these headers are extensible such that further protocols can be built on top; for example, WS-Transactionality, WS-Security, and so on. In short, protocols make it possible to carry out more sophisticated interactions over a transport layer.

Figure 4. Multiple different technical interfaces to a single system.

You need to know: As shown in Figure 4, a single system can offer multiple technical interfaces, each suited to specific needs. It is often easy to see how interfaces are technically different as they have different "protocol, transport, data format" combinations. What is less obvious is that some interfaces might not expose all principal data objects. Both of these aspects are captured in Figure 4.

Notice that REST does not appear as a protocol. This is because REST is a "style" of interaction that is gaining in popularity, but you still need to choose a protocol to implement it; typically, but not necessarily, HTTP. See Interaction type for more detail on style.

Why do you care? Standardised protocols improve re-usability. SOAP is used to provide the message interchange wrappers for most SOA services because it has a well defined structure, and most platforms now have protocol stacks that support it. Unusual provider protocols will, of course, imply the need for a protocol switch integration pattern, adding more complexity to the solution.

Data format

Description: This is the "wire format" of the data. How should the requester transform the data for it to be used in the interface?

Example:
• Is the data carried as text or binary?
• If text, what codepage does it use (for example, UTF-8, Unicode, EBCDIC, ASCII, 7-, 8- or 16-bit, multi-byte characters, and so on)?
• How is the data represented? For example, if text, is it XML, JSON, delimited, CSV?
• If binary, how is its structure defined (for example, COBOL copybook, structures, and so on)?
• What about individual data fields? How is currency formatted (for example, "$123456.00")?
  This could vary by country (for example, "£123,456.89", "€123.456,89"). Notice the different use of commas and dots.
• What about date formats? These vary enormously; for example, "2010-10-17T09:34:22+01:00", "2011.06.08.14.33.56", "Tuesday, 30 August 2011."

You need to know: It is all too easy to make assumptions about the simplicity of a format based on over-simplified sample data. A simple example would be comma separate values (CSV). From a simple example line such as this one showing country codes:

"UK, "GB", "+44"

you might think it is the same as a "delimited" string, but with a comma for the delimiter and quotes around each piece of data. But the reality is much more complex. Take a look at this example representing data in a particular sequence:

"Size 2 Screwdriver", PAID, PART12345678, 24, $5.59, "Mr Smith, West Street, Big City"

Notice the subtleties about the format. For example, if you just used a delimited data handler looking for commas, you would incorrectly break up the address at the end into three separate fields. Also, not all data fields use quotes; here, quotes are only used if there is a space or comma in the text field, and they are never used for numbers. Clearly, parsing CSV data is significantly more complex than delimited data, and this is a very simple example in comparison to the range of formats you could come across in integration solutions.

Why do you care? Text formats XML and JSON are the most readily used formats for re-usable service interfaces. Less common formats will be hard for requestors to parse or create, resulting in the need for data handler integration patterns. The full variation of formats that will need to be handled must be fully understood. Again, you can see the benefit for an SOA to standardise things such as data formats across an enterprise to make services more re-usable.

C. Interaction type

What does the complete interaction between requester and provider look like? A number of characteristics are used to understand how the interaction is performed over the transport. Figure 5 shows the basic interaction types.
The interaction types in (a) and (b) are essentially those provided by raw use of the transport layer, and so they look essentially the same as that shown in Figure 3. However, (c) uses the transport in a more complex way. (You will see even more complex interactions types later.)

a. **Fire and forget, with non-blocking transport**: A messaging transport is used. The request is one-way, sending data to the provider, but with no response. The requester needs only to wait for the message to be received by the transport; that is, it is non-blocking.

b. **Request-response, with blocking transport**: A synchronous transport is used. The requester requires response data and must wait for the provider to respond. The provider must be available at the time of the request and must process the request immediately.

c. **Request-response, with non-blocking transport**: A messaging transport is used. The requester need only wait for the message to be received by the transport; it is then freed up to do other work. It must, however, provide a way in which it can receive messages from a response queue at a later time and be able to correlate this with the original request.

**Request-response or fire-forget**

**Description**: Do you expect a response confirming that the action has occurred, and potentially with returning data (as in (b) and (c) in Figure 5), or are you simply queuing an action to occur at a later time (as in (a) in Figure 5)?

**Example**:

Web services using SOAP/HTTP are typically used for request-response interactions due to the simplistic correlation of request and response messages for the requester, as shown as type (b) in Figure 5. HTTP has the further advantage of needing little configuration to connect the requester and provider.

Fire-forget calls are often done over a transport such as WebSphere MQ so that they can take advantage of the assured delivery provided by the messaging transport to ensure the request is eventually received. However, do not assume that if the transport is messaging it is automatically a fire-forget interaction type. Messaging is equally capable of being used for request-response. In Figure 5, both (a) and (c) use messaging, but they are fire-forget and request-response, respectively.

**You should know**: Sometimes you can derive the answer to this interface characteristic from other characteristics:
• "Read" type requests are by definition request-response, as the purpose of the call is to "read" the response data.
• "Change" type calls could be either style, depending whether they simply lodge a request and respond with an acknowledgement that the request has been received (fire-forget), or perform the full processing of the change before responding (request-response).

Web services can be used to perform a type of fire-forget interaction too, where the response is just a simple acknowledgement and the actual processing happens separately. This is shown in the advanced interaction types as (d) in Figure 6.

**Figure 6. Advanced interaction types**

- **d. Request with acknowledgement, using blocking transport:** A synchronous transport is used. The provider must be present to receive the request, but it provides only an acknowledgement that the action will be performed at a later time. This is similar to fire-forget in that the actual business task required by the requester is performed after the acknowledgement is returned, but this is more tightly coupled.
- **e. Request-response blocking caller API, using non-blocking transport:** This is a hybrid between transport types. A messaging transport is used to communicate with the provider, but the requestor interacts with the messaging using a synchronous API. Many messaging transports provide this type of API to simplify requester interaction with the messaging layer. The provider can process the message at its convenience, but the requester is blocked waiting.
- **f. Request-response by call-back, using blocking transport:** A synchronous transport (such as HTTP) is used to make two separate interactions for request and call-back. Each interaction is individually blocking but releases the block as soon as an acknowledgement is received. This is similar to request-response non-blocking, but the requester and provider interact more directly.

For a significantly broader view of interaction patterns, see *Enterprise Connectivity Patterns: Implementing integration solutions with IBM’s Enterprise Service Bus products*.

**Why do you care?** Request-response calls imply the need to wait for a back end system, or at least correlate with its responses. Each of these has key design implications in terms of expiration, and mechanisms for handling responses. In contrast, fire-forget needs only a swift response from the transport layer, but you have to accept the design consideration that the action you have requested might not yet have happened — indeed, might never happen, and you will not be notified either way.
**Thread-blocking or asynchronous**

**Description:** Are the requesters forced to wait whilst the operation is performed, or does the programming interface enable them to initiate the action and pick up the response separately? What you are interested in is whether the overall interaction is blocking, not just the underlying transport, so this is not as simple as the concept shown in Figure 3.

**Example:** The blocking transport HTTP is the most commonly used transport for request-response SOAP protocol requests; in this situation, it is thread blocking interaction type (b), shown in Figure 5. However, HTTP can be used in other more creative ways, such as interaction types (d) and (f) in Figure 6, where the overall interaction type is non-blocking from the requester’s point of view. Even though the transport itself is briefly blocking the requestor, it does not block the overall interaction.

Messaging-based request/response interactions always have the capacity to be non-locking. However, whether they are locking or not depends on how the requester codes to the messaging API, or indeed which APIs are made available to the requester. Interaction (d) in Figure 6 shows a hybrid, where the requester is genuinely blocked throughout the interaction, despite the fact that a messaging-based non-blocking transport is in use.

**Why do you care?** It can be argued that non-thread blocking interactions can scale more effectively; by definition, they use less thread resources. However, they present complex challenges in terms of correlating the request with response. Error handling can be more complex, as it cannot be performed in a single end to end transaction.

**Batch or individual**

**Description:** Does the interface process individual business actions, or does it take a group of actions at a time and process them together?

**Example:** Early integration between systems was done by extracting a batch file containing data relating to multiple business actions from one system and processing it with batch processing code in another. These batch operations would only occur periodically; for example, daily, at month end, and so on. Many of these interfaces still exist, and indeed in some cases continue to be the most efficient way to process large amounts of data.

Modern interfaces more often work at the level of individual actions, and enable data operations between systems to be performed in real time. The bulk of this section so far has been discussing interfaces built on SOAP/HTTP or WebSphere MQ, and assumed that they process individual events, which is normally the case. However, there will be occasions where these interfaces will actually carry batch loads. As a simple example, take a common example, called submitOrder(Order), which processes a single order from a customer. Imagine if this were instead designed to pass the batch of orders placed by a store so they could be communicated to the main office; for example, submit(OrderList []). In fact, it would more likely be submit(OrderBatch), where OrderBatch contained the OrderList batch, but also various data that applied to the whole batch of orders. Message-based transport is equally often used for handling batches. Message transports are arguably better at handling large batch data objects than transports like HTTP.
Why do you care? Batch processing completely changes the characteristics of the interfaces, from error handling, to transactionality, to performance, and likewise the patterns required and the tools required to perform the work. Specialist tools now exist for handling large volumes of data in blocks, termed Extract Transform Load (ETL), of which IBM InfoSphere® DataStage® is an example. These are very different in character from tools that perform event-by-event processing. If mostly batch work is required, ETL tools should be considered rather than ESB (Enterprise Service Bus) or process-based tools that typically work on individual events. However, it might be appropriate to use ESB capabilities to turn batch interactions into individual actions and vice versa.

Message size

Description: The size of a message that is comfortably accepted by the provider whereby it is still able to meet its service level agreements. Is there an agreed maximum size limit, or at least a size that is considered inappropriate, or are large messages perhaps even blocked or rejected?

Example: Some ways that interfaces manage large objects include:

- Reject all messages over 100K to ensure the service cannot be brought down by malicious "large object" attacks.
- Accept a typical message size of 10kbytes, although test to 1Mb to ensure handling of known but rare cases of larger messages. Greater than 1Mb is rejected.
- Provide a mechanism for adding attachments (like in e-mail, and SOAP) that ensures that large objects can be recognized and handled separately.
- Permit large objects to be sent by "chunking" — breaking up the object into manageable chunks that will be re-assembled on the other side, in order to keep the individual message size down. This enables the object to be processed in parallel if there are sufficient resources available.
- Provide a "streaming" capability whereby the object is sent progressively over an open connection, so the entire object is never present as a single object. This is similar to chunking, but the requester and provider are in relatively direct and continuous communication.

You need to know: When introducing a new requester's requirements, you should consider the adage Just because you can, doesn't mean you should; just because an interface can accept 10Mb messages, it does not mean you should assume that to be the normal usage scenario.

Unless a system is specifically designed to manage large objects, 10Mb of message will have to sit in a system's memory somewhere, and worse, copies will probably be made of it during its passage through the system. Consider the memory management being done by a JVM (Java™ Virtual Machine); if many 10Mb objects are passing through the system it will become increasingly difficult for the memory manager to find continuous slots of space, so the objects will have to be fragmented, slowing down the JVM in much the same way as a hard disk slows down if the files it is storing are fragmented.

It is critical to accept that the new requester is becoming part of a community of existing requesters. They must all play nicely with the interface if they are to continue to get good service. If they have radically different requirements, they should bring these to the table openly, rather than attempting to squeeze them through the existing interface.
Why do you care? A new requester might use an interface in a way that appears to be identical to what has been used before, but sending dramatically larger messages. A common example is to suddenly include images in what before were small documents. This would have a significant effect on memory and CPU in the provider and any integration components. One common alternative to passing large messages end to end is to use the claim check integration pattern, shown in Figure 7, where the large object is persisted and only a reference to the object is passed on the interface.

Figure 7. "Claim Check" pattern

D. Performance

There are two primary uses of performance-related interface characteristics:

- **Governance**: How do you know if the interface is performing to its service level agreement (SLA)? How do you know if a requester is "over-using" an interface? You need to be able to monitor usage, you need thresholds against which to assess the monitoring information, and you need to know who to alert, should the thresholds be over-stepped.

- **Capacity planning**: As it stands, does the interface have the ability to scale as new requesters are introduced? You need to be able to predict the future usage of the interface, and provide sufficient warning when capacity needs to be increased.

The performance-related characteristics provide you with essential non-functional measures, the restrictions of which would not otherwise be immediately obvious during the early analysis phase.

Response times

**Description**: The duration between a request made and the receipt of a response time via this interface.

**Example**: This is typically expressed as an average response time in units such as requests per second, as shown in the left side of Figure 8.
Figure 8. Sequence diagram showing response time and throughput

You need to know: Ideally, response time figures would be provided with some idea of the range and a concurrency. These are not easy to show on a sequence diagram, but let’s describe them briefly nonetheless.

The range (or spread) of response times is at least as important as the value for the average. It is particularly important to consider the spread or profile of response times, not just the average. For example, if the average is 5 seconds, and most requests are within 2-7 seconds, that is very different than if the average is 5 because occasionally the response takes up to 50 seconds, but the rest of the time it is less than 1 second.

Concurrency itself is a separate characteristic discussed later, but note that sequence diagrams do not represent concurrency and parallel processing very well. The example in Figure 7 implies only one request can happen at a time. However, most interfaces permit concurrent requests, and this would have to be taken into account when measuring the response time and throughput figures. For example, you might say an interface provides these response time characteristics:

Average response time = 3 seconds, 95% of the time, for 10 concurrent requests.

Why do you care? Requesters might have expectations in terms of GUI (graphical user interface) responsiveness, or limitations on threads, or transaction timeouts, that would be adversely affected if response times were slow.

Throughput
Description: The number of requests that can be processed over a relevant period of time.

Example: The right side of Figure 8 shows an example where the throughput looks to be roughly 30 requests per minute. As was noted under response time, this assumes that the provider does not offer parallelism, as this would be hard to show on a sequence diagram. (See the section on concurrency for more.) Other time units are equally common, such as 10,000 records per hour, 10 requests per second, and so on.

You need to know: Whilst throughput is typically measured in number of requests over a time period, this can be misleading, as what you really want to know is the sustained (or continuous)
throughput capacity. You might, for example, be able to make 50 requests to an interface within 2 seconds. Is the throughput therefore 25 requests per second? That might be an unsustainable workload, and the provider may have had to queue up the requests internally. It might be 5 seconds before they are all complete. Rather than thinking of it as the request rate, you might get a more realistic figure if you think of it more as the response rate. In this case, 50 responses in 5 seconds is more realistic than 10 per second.

The period of time should be relevant to the usage and type of the interface. An interface used by a human user in real time might need a requests per second measurement, whereas a file-based interaction might be measured in the number of records within a file that could be read and processed within an hour.

Remember that 6000 request per hour does not necessarily mean 100 per minute. You need to know when the peak periods are (for example, opening hour, lunch time, and so on). Is your system permitted to "catch up" later, or do you always need to be on top of the load?

**Why do you care?** Just because you are able to call a fire-forget interface at 100 requests per second does not mean the provider can actually process them at this rate. They could simply be building up in a queue. On request-response interfaces, if you exceed the throughput for a sustained period, the provider might run out of resources, at best resulting in failure responses, and at worst bringing down the provider completely; the latter is what happens in a "denial of service" attack. If you want predictable behaviour from your interfaces, you need to correctly understand their capacity.

**Volumes**

**Description:** The aggregate volumes across longer periods of time. Volumes are a measure of maximum capacity over a long enough interval to include aberrations due to maintenance, outages, and periodic variations in capacity where resources are shared.

**Example:** Volumes would be measured at least across days (for example, 10000 records per day), perhaps even over months or a year (for example, 1 million records per month). You need to look at the profile of the load, sometimes called **shaping**, whereby, for example, you might be able to handle 1000 requests per minute during the day, but only 50 per minute at night due to the CPU taken up by batch processing.

**You need to know:** This measure is more often captured when looking at the requesters' requirements in order to establish whether the provider can meet the overall demand. A maximum capacity of the provider is derivable by combining the availability with the throughput.

**Why do you care?** Volumes are essential in capacity planning and provide an immediate understanding of whether new requesters' requirements are achievable with the existing infrastructure. Resolution could be as simple as adding CPU to providers or bandwidth to a network. However, it could mean a much more fundamental refactoring of the architecture, which you would need to be aware of as early as possible.
Concurrency

**Description:** The number of requests that can be processed at the same time.

**Example:** Figure 9 shows how a provider might be processing multiple transactions at once in parallel. In this case, you have a concurrency of 3.

**Figure 9. Concurrent/parallel transactions over time**

This might be a hard limit constrained by the number of connections you can initiate over the interface, as would be the case with JDBC, or it could be a soft limit whereby the provider continues to accept new requests, but performance significantly degrades when you go beyond the number of concurrent threads available for processing in the provider.

Here are two very different interfaces and how concurrency would be defined for each:

- A JDBC interface allows no more than 10 concurrent requests to be made to the provider over the interface. Additional requests will be rejected.
- A WebSphere MQ-based interface permits a total of 1000 requests based on the maximum queue depth, but the provider only has threads to process 20 at a time. Here, you have a maximum of concurrent requests of 1000, but the level of sustainable concurrent requests is only 20.

**You should know:** In the second case, you see there is a stark difference between maximum concurrency and sustainable concurrency. You would be most interested in sustainable concurrency for throughput-based use of the interface, whereas you would be interested in the profile of the response time for response time based scenarios, even beyond the sustainable concurrency, on the basis that you might have brief peaks in load and would want to know if the response time remains acceptable.

**Why do you care?** If your requester requires higher concurrency than the provider currently offers, you might need to introduce a **store and forward** pattern to queue incoming requests and give the impression to a caller that a higher concurrency is available, as in the second example above. This can simplify error handling for the caller, although it might result in apparently slower response times when significant numbers of requests are queued, and actually increase error handling complexity if responses are unreasonably delayed. Conversely, if a guaranteed low response time is more important than the occasional error, then a concurrency limiter might need to be implemented to ensure a limited resource is never overloaded.

You might also need to look at load balancing in order to spread requests across available resources in order to achieve the desired concurrency. If you are doing composition, you might
need to use some concurrency to do parallel patterns such as scatter/gather to aggregate multiple requests in a reasonable period of time. Can the available concurrency cope with this extra load?

**E. Integrity**

**Validation**

**Description:** What data validation rules must a requester adhere to for a request to be accepted?

**Example:** Mandatory fields, data format of specific fields, inter-field validation rules, enumerated values, references/identifiers/keys that must exist, and so on.

**Why do you care?** Ideally, validation is performed as part of a request-response interaction pattern to enable the requester to be informed of these known exception conditions and handle the issue immediately.

If the interface is fire-forget, then validation can only occur after a request has been received. In this situation, it will be particularly important for requesters to be aware of the validation rules and to understand that their request could fail even through it has apparently been accepted.

It might be necessary to add a pre-validation to the request in the integration layer to ensure that invalid data is captured whilst the requestor is still present.

Batch fire-forget interfaces (such as EDIFACT and interfaces such as BACS) also have complex validation rules. In both fire-forget cases there will be a need for an offline error handling pattern that could involve a complex separate interaction pattern in its own right.

**Transactionality**

**Description:** What level of transactional support does the interface provide? The term **transactional** means that a set of actions grouped under a transaction as a unit of work is always either done completely or not at all, nothing in between. Transactions should comply with four primary requirements known by the mnemonic ACID:

- **Atomicity:** The actions performed by the transaction are done as a whole or not at all.
- **Consistency:** The transaction must work only with consistent information.
- **Isolation:** The processes coming from two or more transactions must be isolated from one another.
- **Durability:** The changes made by the transaction must be permanent.

This is a well understood and documented topic so there is no need to expand much further on it here. However, although this is a relatively simple concept, there is some significant detail in understanding the exact scope of the transactions, especially when you take into account the different permutations of interaction style, protocol, and transport.

**Example:** The most commonly used transactional system is a database. As such, interface standards such as JDBC enable transactional operations to be controlled by a requester. This is typical of a transactional protocol over a synchronous medium, as shown in example (a) of Figure 10.
Figure 10. Transactions across a request response interface using a synchronous transport

An alternative example would be a typical interaction over web services as SOAP/HTTP. In common usage, the SOAP/HTTP protocol is not transactional, so the transactional scopes look like those in example (b) of Figure 10. It is possible to introduce transactionality with supplementary standards such as WS-Transactionality, but this is rarely desirable since the circumstances where web services are used (such as to expose re-usable services within a service oriented architecture) are usually where it is desirable to have very loose coupling between requestor and provider.

Figure 11 shows that for fire-forget interactions, at least two transactions are involved. One initiated by the requester to put the message onto the queue, and another initiated by the provider to pick it up and process it.

Figure 11. Transactions across a fire-forget interface using a messaging transport

By the time you look at request-response over a messaging transport, you can see from Figure 12 that things are becoming rather more complex. The minimum is three transactions, and often the provider will break up Transaction 2 even further. It is worth considering the error handling here. What will the requester do, having committed Transaction 1, if it does not receive a response from the provider within a reasonable amount of time? It cannot know whether the action it requested has been performed or not. That action could be held up briefly due to a temporary system outage, or there could be a more serious problem and cannot therefore be processed, such as missing data in the request. How can requesters require confirmation that the action has been performed before continuing to the next step, should they want to?
As an exercise, you might like to take what was discussed on Figure 12 and consider what additional complexities are introduced with the hybrid interaction in Figure 6 example (b).

You should know: To say an interface is transactional can mean a number of different things:

- **Internally transactional**: A request results in a single unit of work within the provider. If you get an error back from a system which is not internally transactional, how do you know if the call left the data in a consistent state or not? With modern systems, at least this level of transactionality can usually be assumed, but it is still worth asking the question if the back end technology is an unknown.

- **Callable transactionally**: Transaction with a single system can be controlled by the caller, potentially combined with other requests to the same system, and the commit/rollback choice is made by the requester. This requires transactional protocol such as JDBC. This is the minimum required for example (a) in Figure 10.

- **Potential global transaction participant**: Can participate in two phase commit transaction across multiple providers. The providers and protocol must support XA-compliance. This is the minimum requirement if an interaction such as example (a) in Figure 10 must be combined in a transaction that also includes a state change within the requester or with other back end systems. An example of this would be an automated process engine, making a request to one or more back end systems.

- **Transactionally queued**: Request transactionally puts work in a queue for subsequent processing in a separate transaction. This is, of course, typical of interactions over a messaging transport. Using messaging transports, among other advantages, releases the requester from needing to establish transactional connections with providers. They need only now how to transactionally communicate with the messaging transport. However, it introduces complexities, as discussed above.

Generally, transactionality is only relevant on operations that change data. Read operations do not generally require (or care about) transactionality, although there are some subtle cases where they do, such as a "read for update" where records are deliberately locked using a read operation, so that they stay in a consistent state throughout a subsequent update. Another example is read audits, as described in the privacy section.
**Why do you care?** It should be clear that an understanding of the transactional behavior is essential when understanding full the complexity of implementation. If an interface cannot provide an appropriately transactional interface for the requester's purposes, there is a risk that a failure could result in data being left in an inconsistent state. More complex error handling integration patterns might be required, such as compensation.

There is an opposing view also. Transactionality over an interface nearly always results in some locking of the provider's resources. A transactional interface means the requester has control over how long the locks are held open. This might be an inappropriate level of coupling, especially if the requesters are largely unknown to the provider. You, therefore, can deliberately choose not to enable participation in a global transaction to reduce inappropriate coupling, and manage errors with knowledge of this strategy.

**Statefulness**

**Description:** Some form of requester-specific contextual state setup that is required to successfully perform a sequence of calls.

**Example:** JDBC interfaces are common example of a stateful interface. The JDBC connection itself must be explicitly set up, and there is thereby a stateful connection context retained between transactions if you are to avoid the expensive overhead of creating new connections each time. Furthermore, all transactions have to be explicitly started and ended (committed), so even the simplest action requires at least three separate requests over the connection, throughout which a stateful transaction context is retained. In application containers such as Java EE, this connection and transaction management can be handled by the container, and as such might not be visible to the coder; it could be argued that an overall transaction that is completely managed by the container is not in itself stateful, even though the individual requests that make it up are. However, the statefulness of the significant connection session context remains.

Part of the transactional state held during JDBC transactions relates to locks on the underlying data. Any interaction resulting in holding open locks on data across requests, or indeed locks on any underlying resources, should be considered stateful.

By contrast, a typical simple web service request is generally not stateful. No continuous connection or context is held by any intermediary across multiple invocations between requester and provider. Each request occurs completely independently. If any form of connection state is held, it is at a very low level, and is completely invisible to the requester.

**You need to know:** It is common to wrap multiple requests to a database in a single database stored procedure. One of the reasons for this practice is to reduce the number of stateful interactions. There are other reasons of course, including better performance, and keeping data specific logic in the hands of the data owners.

**Why do you care?** The main disadvantage of a stateful interface is that it is tightly coupled to the workings of the provider and tends to be much more complex for a requester to use. Also, stateful interfaces can often be used in a way that is wasteful (whether accidentally or not) of the provider's resources; for example, by holding connections or transactions open for unnecessarily
long periods. Stateful interfaces also cause issues for scalability, since the requester needs to make all calls via a path that has awareness of the session context, which will necessitate some form of session state replication or propagation. One way of resolving this is to force connections to be "sticky," returning to the same endpoint for all requests in the session, but this again increases coupling and sacrifices availability. It is often wise to wrap up the dependent calls using composition, such that the requester sees only a single, coarser grained, request. For all the reasons stated, it is considered particularly bad practice for services to be stateful in an SOA where decoupling and scalability are paramount.

There are advantages to stateful interfaces, however. Due to the context held on both sides of the interchange, less information needs to be passed in each request, so performance is inherently higher. Performing multiple updates consistently is also easier since the context could be in the form of pessimistic locks on the data to ensure the data does not change between requests. Sometimes there are compromises that make sense in the context of a particular design, such as the loss of scalability providing sufficient reward in terms of simplicity of error handling.

**Event sequence**

**Description:** Must requests be acted upon in the same order as they are received?

**Example:** A customer create request must not be overtaken by a customer update for the same customer, since the update will fail with a customer not found exception. In Figure 13, for example, messages containing business actions to "create a customer" and "add an address to a customer" are placed in a queue. To improve performance, multiple threads of execution are permitted to draw work from the queue so that actions can be performed simultaneously by separate threads.

**Figure 13: Example of an error caused by loss of event sequence**

If both your customer actions are picked up by different threads and worked on in parallel, it is likely that the action to add a customer address will fail because the customer has not yet been created. The actions (events) have become out of sequence.

**You should know:** Event sequence is typically only relevant to non thread-blocking transports that are often queue based, where there is risk that messages could overtake one another. Synchronous thread blocking transports typically used when making requests from user interfaces usually preserve order naturally, as requestors can only make one call at a time anyway. However, with the introduction of more user interfaces that enable users to do multiple things in parallel (using technologies such as Ajax), event sequence will become even more relevant.
**Why do you care?** If order is critical, you will need to ensure sequence is retained. There are a variety of patterns to achieve this, from deliberately single threading related requests to introducing re-sequencer patterns.

**Idempotence**

**Description:** If the same request is made twice, will it achieve the same effect, or will it have an additional effect, such as duplicates? If an interface is idempotent, then two or more identical requests should result in the same effect as the first one.

**Example:** Assume you have an operation on an interface that enables you to submit a mortgage application. What would happen if you called the operation twice in succession with the same data? Would it:

- Result in a duplicate mortgage application? If so, then the interface is not idempotent, and requesters would have to be aware of this risk. For this example, this is probably an inappropriate behavior for the interface, as two mortgage applications for the same data would not make business sense.

- Result in an error (duplicate submission) on the second request? In this case, the operation would be functionally idempotent. You need to consider how you detected the duplicates. Was there some primary data within the application that you compared across the two requests (for example, applicant name and date of birth)? If so, what if there were different data in the remainder of the request? How should you handle that? The error message above would no longer be entirely correct.

- Result in success on both invocations, but only the first application submission is actually processed? In this case, the operation would be both functionally and behaviorally idempotent. In this scenario, it would only be valid to behave in this way if the requests were found to be completely identical, otherwise you could mislead the user into thinking that different data in the second request had actually been processed.

In the above example, it is obvious that duplicate submissions must be in error. However, this is clearly not always the case. Imagine ordering books via an online bookstore. An administrator buying books on behalf of students might legitimately make multiple identical purchases in a short span of time.

**You should know:** If you are to be able to enforce true transactionality all the way from requester to provider, you generally need not concern yourself with duplicates from re-tries, but as noted in the transactionality section, this is not always possible, or indeed desirable. If there are any non-transactional hops along the way, then requesters can receive errors that leave the success of the interaction "in-doubt" and re-tries could cause duplicates for non-idempotent interfaces.

**Why do you care?** Knowledge of the idempotence of an interface is critical in order to put in place appropriate error handling for the requester, or indeed the integration layer to ensure you get the correct result when multiple identical requests are made.

Idempotence is particularly relevant in SOA, as web services are typically not set up to be transactional. It is also a common risk with request-response queue-based interactions. If a
response is not received, how can the requester know whether the request was received by the provider or not? Even if it was received, was it processed correctly? Managing idempotence quickly becomes non-trivial.

If duplicate requests would be a problem, you need to introduce idempotence either in the provider or in the integration layer. There a number of techniques for this which generally rely on proving the uniqueness of a request. For example:

- **Unique data**: A transaction ID that can only be used once.
- **Time span**: Two payments for the same annual insurance premium within the same day would likely be duplicates.

**F. Security**

Security considerations are one of the most commonly overlooked aspects of interface requirements. Functionally, an interface can appear to work perfectly in simpler environments, and a project could progress in ignorance of the complexity that lies ahead. Retrofitting security can be extremely complex, requiring anything from re-building of infrastructures and environments, redesigning, refactoring and re-testing of extraordinary amounts of code. You might well find that an interface that is perfect in every other respect is untenable from a security point of view, which could result in the creation of a brand new interface at a later stage.

**Identity/authentication**

**Description**: Identities are critical for ensuring that only agreed users and systems are able to make requests, establishing authorization (more on this in the next section), and for auditing actions performed. There are two key types of identity relevant to interfaces: requesting user identity and requesting system identity. Interfaces can require one, both, or neither of these.

**Example:**

**Requesting user identity**: How the interface recognizes "who" is performing the action. For example:

- Identity is known to the session/connection, as in basic HTTP authentication or JDBC connections.
- Identity is carried by the protocol (for example, SOAP WS-Security) as a token (for example, SAML or LTPA).
- Identity is carried in specific fields in the body of the message, as is typically the case with a home-grown security mechanism.

**Requesting system identity**: Is the interface aware of the identity of the calling system? For example:

- Explicitly configured certificate authenticated HTTPS channels between systems.
- Calling system identifier carried in headers or body of the message.
- Firewall controlled IP access between systems; you might not know the identity of the system, but you know it can only have come from a machine you trust.
**You should know:** How the identity was authenticated as being the user/system that it claims to be is usually beyond scope of the interface itself, but it is a critical part of any validation of the broader security architecture of the solution.

**Why do you care?** You must ensure that the calling system has the relevant identities at its disposal when making requests, and the transport protocol and any integration layer must be able to propagate these identities via a trusted mechanism.

These identities need to be the ones relevant to the security domain of the provider. In simplistic situations, you might be able to effectively hardcode a single user ID for all requests, but this could be misleading in the audit trail. If so, more sophisticated mechanisms of federated identity management will need to be considered to translate user identities on the requesting system’s domain to users on the provider’s domain.

**Authorization**

**Description:** How the interface establishes whether an identity has the authority to perform the requested action.

**Example:** In simple cases, authorization might be hardcoded into the requesting application such that it can only ever make requests that it is permitted to make. This is obviously not ideal from a security point of view, and it is completely inappropriate in a broader re-use scenario such as SOA.

Authorization mechanisms are often specific to the system being called, and as such might be reliant on the identity propagated being used compared to the roles it is permitted to perform. An example of this would be the authorization present in Java EE systems such as that on EJB methods.

An alternative is to delegate authorization to a separate component, such as IBM Tivoli® Access Manager. This clearly has the advantage enabling authorizations to be managed centrally. However, as with any centralization, this means more skills and teams are then required to deploy the application.

**You should know:** Authorization is typically established via roles and groups, as administering at the level of individual users and operations would be unmanageable.

**Why do you care?** Usage of the interface in the past might have been through a single trusted application that performed authorization on behalf of the provider. If you now make this interface available to other requesters, you have to trust each requester to write appropriate authorization and access control logic into their application. This is generally an unacceptable risk and adds far too much complexity to the requesting applications. Alternatives typically mean extracting authorization logic into the integration layer, or better still, into an independent access management component which the integration layer can call.

You might also have to consider "adopted authority" patterns; for example, the provider trusts the requester, so that when the requester says "do this on behalf of user A," the provider executes the request, without making a second check on who user A actually is.
Data ownership

Description: Who owns the data touched by the operation? Which system holds the master of the data?

Example: Some examples of where the data can be owned might be:

- The system providing this interface owns the data. Reads and changes to data can be performed by users who are suitably authorized.
- The data in the system are a replica. Through this interface data can be read, but not updated.
- The data are not the master, but changes to them can be made and are replicated automatically up to the master. Here, risk of conflicting updates need to be considered.

Why do you care? The existence of an interface on a provider that permits you to change data does not in any way imply it is the most appropriate way to update that data from the broader point of view of the enterprise. It is very common in large enterprises to find that important data entities are actually duplicated into multiple separate systems. "Customer" data is commonly spread across customer relationship management systems, billing systems, marketing systems, and many others. Each system might contain different pieces of the customer data. Consider the complexity of an apparently simple change of address operation under these circumstances. You need to consider whether you are updating the data through the right interface, whether there is a data management strategy in place that you must adhere to, and whether you should be introducing a data synchronization pattern to ensure all sources of the data are kept up to date.

Privacy

Description: Does it matter who can see the data that is sent or received via the interface?

Example: There are different types of privacy issues. Some of the most common are:

- Data authorization: Salesmen should only be able to see the data in their region.
- Data encryption: Credit card details must not be visible in transit.
- Digital signatures: You must be able to prove that large orders were made by the requester before processing them.
- Read auditing: You might need to record who has performed reads on the data, so that you know who has formally seen it.

Why do you care? Authorization (discussed earlier) provides some level of privacy, but only acts at the operations level, not at the data level. This is especially true where specific attributes of data need to be treated in a special manner. For example, it could be that all credit card numbers need to be encrypted or they might need to be removed entirely from any logging performed by the integration layer. Data privacy patterns are by nature very specific to the business data, so this can rarely be accomplished in a generic way, and will usually require explicit, sometimes complex, and often CPU-intensive code. The need for these patterns will have a significant impact on implementation estimates, and possibly software license costs.
G. Reliability

Availability

Description: When will the interface definitely be available? How often and for how long do unexpected outages occur?

Example: Availability is one of the most important measures within a service level agreement, and as such there are some fairly common ways of measuring it, such as:

• **High availability:** The percentage availability of a system is often measured in "nines," based on the ratio between the down time and the total intended availability. For example, 99.9% availability is "three nines" availability and works out to an average of about 10 minutes per day of outages.
• **Availability windows:** When can the system be called via this interface? Are there scheduled downtimes for maintenance? Is it only supported during office hours? For example, 24/7, 9:00 am – 5:00 pm weekdays, 5:00 am -11:00 pm excluding bank holidays, and so on.
• **Mean time between failures:** What is the average time between outages? This metric is more often used as a way of retrospectively assessing whether the SLA is being achieved.

**Why do you care?** A provider with less availability than required could be a catastrophic problem, no matter how many other characteristics match. This is typically a challenging problem to resolve as availability — especially of an aging system — can be very expensive to improve on the system itself. If the requester requires greater availability than the provider offers, you might need to consider changing some request-response interfaces to fire-forget or introduce store and forward interaction patterns.

Delivery assurance

Description: If a request has been made, can you have complete confidence that it will be processed?

Example:

• If requests are made over non-transactional protocols (which is the default for web services) and a communication failure occurs between the request and the response, it would be impossible to know whether the request was actually received or not. Web service requests without WS-Transactionality do not, therefore, have delivery assurance.
• If the interface uses a transactional message-oriented transport, such as WebSphere MQ, where the messages are persisted (as opposed to in memory) and the provider transactionally adds the message to the queue, you have delivery assurance. Of course, to be completely assured you would also have to consider the disaster recovery strategy, such as whether the queue data is mirrored to a second data centre.

You should know: There is a further aspect to assure delivery: once only delivery. Can you be sure that the message will only be processed once? If the provider, for example, does not itself pick up messages within a transaction, under certain error conditions it could process a message twice or more.
Why do you care? If your business transactions have either a high value or would have high cost side effects when they are lost, you will need to ensure that the interface has suitably robust transport, transactionality, and recovery in place. If the provider does not offer the required level of assurance, you might need to introduce an assured delivery pattern through the integration layer.

H. Error handling

Error management capabilities

Description: If errors occur when invoking the interface, how are they handled, and by whom?

Example: You are really interested in how the error is propagated along the interface so you can see who is affected by it and who must react to it.

- In blocking transports, such as HTTP, errors must typically be pushed immediately back to the requesters, so that they can free up the blocked thread. The requester is then expected to manage any resubmission. The error might also be noted in the logs so that, for example, they could be analysed on a daily basis for frequently occurring issues.
- For interfaces provided over non-blocking transports, other error management options are possible. Whilst errors could be reflected directly back to the requester, it might be more appropriate to hide some errors from the caller by storing the error and notifying support staff. If the issue is resolved, a successful response is sent and the requester does not need to know there was an issue.

Why do you care? The error handling strategy employed by the provider must be appropriate for the requester. For example, can the requester manage error resubmission? How can the requester find out if a request that has been in progress for a long time is stalled pending intervention from support?

Known exception conditions

Description: Errors that you have an understanding of at design time, and therefore can expect to be meaningfully reported back to the caller.

Example: There are two primary types of errors that could be reported back to the requester in a meaningful way:

- **Business errors:** Errors that mean something in the functional context of the interface. For example, "Customer not found" (on search), "Customer already exists" (on create), "No delivery slots available on requested day."
- **System errors:** Errors that have nothing to do with the functional action performed by the interface, but relate to aspects such as the provider availability or network status. For example, "System unavailable due to scheduled maintenance," "Request timed out due to overloaded system," "No connections available."

Why do you care? The requester will need to know what business errors are surfaced, as there might be a need for specific code in the requester to manage these situations. For example, a
"customer not found" error might mean taking a user back to a different page to check that the customer details have been entered correctly.

Some system errors might be transient, in which case you might wish to implement retry logic into the integration layer so they are less commonly seen by the requester.

**Unexpected error presentation**

**Description:** How unexpected errors will be presented to the requester.

**Example:** Providers vary in how well behaved they are when things go wrong:

- Unknown errors are passed back as a SOAP fault with a specific fault code. This is a well-controlled unknown error presentation.
- Unknown errors are sent back as free text message with un-specified structure. This is more difficult for the requester to manage, but at least predictable.
- Unknown errors result in a variety of behaviors, from HTTP lost connections, to HTTP error responses, to free text HTTP responses. The request might even hang indefinitely. This is a challenging scenario, which is very difficult for a requester to manage.

**Why do you care?** You need to ascertain what level of sophistication your requesters will require to be able to catch, control, and log unexpected errors when they occur. You will need to wrap requests in a complex error handler, or even an isolated adapter to ensure these unexpected errors do not affect the requester's runtime.

**Conclusion**

It should now be clear why, as stated in the introduction, that while integration protocols, styles, and architectures have changed and evolved, the fundamental characteristics of integration between systems has remained largely the same. All of the characteristics here have been relevant since computer systems first began talking to one another, and will continue to be essential for the foreseeable future.

You now have a complete, detailed picture of what each of the integration characteristics represents, why it is important, and what needs to be captured for successful and reliable integration. You can combine this information with that from [Part 1](#), which explained when these various characteristics should be captured in the lifecycle of a project.

Armed with this knowledge, you should be able to assess the complexity of integration requirements for projects in a more systematic way, reducing project risk, and improving the quality of the final technical solution.

**Acknowledgements**

The authors would like to thank the following Andy Garratt, David George, Geoff Hambrick, Carlo Marcoli, and Claudio Tagliabue for their contributions to and reviews of this article.
Related topics

- Capturing and analyzing interface characteristics, Part 1: Capturing integration complexity for BPM and SOA solutions
- Patterns and Best Practices for Enterprise Integration
- Enterprise Service Bus, re-examined
- Enterprise Connectivity Patterns: Implementing integration solutions with IBM's Enterprise Service Bus products

© Copyright IBM Corporation 2012

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)