Capturing and analyzing interface characteristics, Part 1: Capturing integration complexity for BPM and SOA solutions

Kim J. Clark
Brian M. Petrini

December 07, 2011

This two-part article describes a well-tested technique for capturing the fundamental interface characteristics of integration points with back end systems. Integration still accounts for a majority of the design and implementation effort of many IT solutions. Poorly understood integration requirements represent a significant risk to a project. You will see how correct analysis of the integration points improves estimating and technology selection, and ultimately enables a more predictable pattern-based approach to solution design and implementation. Moreover, these articles will specifically highlight the relevance of this technique to enterprise architecture initiatives, such as service-oriented architecture (SOA) and business process management (BPM).

View more content in this series

Introduction

In terms of importance, one of the most commonly underestimated areas of complexity in an IT solution is the integration with back end systems. Despite the efforts of service-oriented architecture (SOA) to standardize and simplify the way we access back end systems, every new project inevitably brings new points of integration, or, at the very least, enhancements to existing integrations.

This two-part article introduces a way to capture and analyze integration complexity using interface characteristics to improve your ability to plan, design, and ultimately implement solutions involving integration.

Part 1 includes:

- **Introducing interface characteristics**: A look at the breadth of information you need to know to truly understand the complexity of an interface.
- **Integration in SOA and BPM**: Why integration is so important to architectural initiatives such as SOA and BPM.
• **Iterative interface analysis:** Because it would be impractical to capture everything about all the interfaces in a solution in one go, this section discusses which characteristics to capture at which stage in the project, and why.

**Part 2** will dive into each of the individual interface characteristics in detail.

### Introducing interface characteristics

With the tried and tested approach described in these articles for systematically analyzing the integration requirements and capabilities of back end systems, you'll be able to ask just enough of the right questions at each analysis and design stage to ensure that you can effectively estimate, identify risks, and ultimately choose the right design patterns to achieve the desired integration in the solution.

This technique centers on iteratively capturing the key interface characteristics that have the greatest effect on design and implementation. In the simple case of a requester connecting directly to a provider, the interface characteristics are the characteristics of the provider, as shown in Figure 1.

**Figure 1. Summary topics of interface characteristics of a provider**

![Diagram of interface characteristics](image)

Interface characteristics have been presented in summary at many international conferences, and gradually honed on client projects for many years, but this is the first time they have been fully and publicly documented. The full set of key characteristics is shown in Table 1 (these will be described in detail in **Part 2**).

### Table 1. The core set of interface characteristics

<table>
<thead>
<tr>
<th>DATA</th>
<th>INTEGRITY</th>
<th>TECHNICAL INTERFACE</th>
<th>SECURITY</th>
<th>RELIABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal data objects</td>
<td>Validation</td>
<td>Transport</td>
<td>Identity/authentication</td>
<td>Availability</td>
</tr>
<tr>
<td>Operation/function</td>
<td>Transactionality</td>
<td>Protocol</td>
<td>Authorization</td>
<td>Delivery assurance</td>
</tr>
<tr>
<td>Read or change</td>
<td>Statefulness</td>
<td>Data format</td>
<td>Data ownership</td>
<td></td>
</tr>
<tr>
<td>Request/response objects</td>
<td>Event sequence</td>
<td></td>
<td>Privacy</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Capturing and analyzing interface characteristics, Part 1: Capturing integration complexity for BPM and SOA solutions
Hopefully, the scope of these characteristics makes it clear right away why it is that many projects fail to assess integration effectively. It is no doubt also clear that it would take significant experience to capture and assess such a large amount of information at one time. Later on, you’ll see how this can all be broken down into manageable steps.

Figure 2 shows that there are really two sets of interface characteristics to be captured. The set on the right includes the capabilities of the provider, but you also need to know the requirements of the requester.

**Figure 2. Comparing the gap between the interface characteristics of requester and provider**

As you compare the characteristics of requestor and provider, you can then establish the integration patterns that will be required to resolve the differences, as shown in Figure 3.
Integration in SOA and BPM

Few projects are executed in complete isolation. Every project is nearly always part of a broader enterprise initiative. Two common initiatives in recent years have been service-oriented architecture (SOA) and business process management (BPM). It is important to understand why interface characteristics are critical to the broader work and how they can be relevantly applied.

SOA

A look at then and now

If you'd like a review, this earlier article on solution design includes a detailed description of how integration architectures have evolved over time.

SOA is a vast topic, but the aspect of it that applies here is where SOA is seen as a logical extension of traditional integration; taking existing integration techniques and then going a step further to make them truly re-usable across a broader set of requesters. The important difference for this discussion is that traditional integration caters to a known set of requestors for which you perform explicit integration. This is shown in the upper example in Figure 4. You do not, therefore, consider the needs of future potential requestors. Patterns such as "hub and spoke" make it easier to introduce new requesters, but each one still requires explicit integration. SOA goes a step further by aiming to assess what the most useful interfaces will be and expose them as services by standardizing they way they are discovered, as shown in the lower part of Figure 4. The objective is that future requesters can simply "use" the service rather than having to "integrate" with it.
Now, let's take a look at just how important it is to understand the interfaces in this situation.

To expose services effectively, you need to collate interface characteristics from the anticipated requesters for your system (a and b in Figure 4) and also estimate the potential future requesters (c). You must then compare that with the available interfaces available on providers (e). The hardest part comes next, when you have to use all that information and define the idealized "service" that you could expose for re-use. This is shown in Figure 4 as service exposure characteristics (d). These characteristics for your purposes can be thought of as essentially the same as interface characteristics, except that they are for an exposed service. There are differences, mostly in the sense that many of the characteristics are defined by the governance policies of the SOA, but that detail isn't important at this level.

Now, imagine if you did that exercise without all of these interface characteristics. You could get a long way into your design believing that, at a high level, you had a workable solution, only to discover during implementation that some fundamental characteristic completely negates the re-use potential of your service.

Of course, you might not be at the beginning of an SOA initiative. Many services could have already been exposed. If this is the case, you might be able to make some simpler assumptions about how easy it is for your requesters to connect to providers, but you should test our assumptions first. One way to assess this is to understand how mature the SOA is in relation to the service integration maturity model (SIMM) as described by the article referred to above. For example, if interfaces expose SOA-based "governed services," then the enterprise has reached SIMM level 4 (for these interfaces at least) and so you can assume that these interfaces should be easy to re-use. However, do not underestimate just how difficult it is to provide well-governed services. It is almost certain that some of the interfaces will be at a lower level, so it is these that you will need to concentrate on in later phases.
To understand what is meant by governed services, see this article, which re-examines terminology and concepts relating to the enterprise service bus (ESB).

Equally important, just because an SOA initiative is taking place, do not assume that all interfaces need to become fully exposed governed services; this would typically be too expensive for most projects. Each will need to be evaluated for its opportunity for re-use. One mechanism you can use to perform this evaluation is the SOMA Litmus Test (see Related topics).

**BPM**

As with SOA, BPM is also a broad topic, but the aspect of BPM that applies here is how business processes interact with back end systems, or providers.

Figure 5 shows that an end to end business process can interact with multiple systems in many different ways. There are a number of things to notice in this diagram.

First, because the process should not, ideally, need to know the details of how to integrate with each of the back end systems individually, the diagram shows only a logical service layer, hiding the detail of the actual integration necessary to get to the back end systems. Therefore, SOA can clearly be complementary to BPM, making integration points needed by the process more easily available.

Second, notice how the type of service requester changes regularly throughout the process. Requests can come from a graphical user interface, from within a brief composition, from a long lived asynchronous process, and so on. Each of these requesters prefers different integration characteristics in the services it calls.

**Figure 5. The service interactions taking place during an end to end process**

Third, notice how "preferred" integration characteristics vary for different process implementation types. Let's look at just a selection of the interface characteristic across these interactions, specifically around interaction type and transactionality (the numbers in the list below relate to those that appear in Figure 5):

1. Synchronous request-response non-transactional read
2. Synchronous request-response update non-transactional
3. Synchronous request-response read with transaction lock
4. Synchronous request-response with transactional update
5. Asynchronous fire-forget event
6. Asynchronous event with correlation data
7. Asynchronous event receipt with correlation data
8. Asynchronous request-response update

You can see that each interaction has a preference for certain characteristics. For example:

• A flow through a graphical user interface uses mostly synchronous interactions and does not generally expect to be able to transactionally combine them.
• A synchronous transactional composition might require services that could participate in a global transaction.
• A long running process saving state over a significant time period might find it easier to interact with asynchronously-exposed services, and will also be comfortable interacting in an event-based style where data is received via completely separate events that contain correlating data.

So, what should you take from this with regard to the relevance of interface characteristics in relation to BPM? Primarily, you should be aware that for what might appear to be the same interaction, the preferred interface characteristics vary considerably depending on the context in which the interface is used. By establishing the most common process implementation types, you might be able to significantly improve the amount of re-use you gain from services you expose by ensuring they exhibit the right characteristics for the common contextual uses.

One final comment on Figure 5, and on BPM in general. Figure 5 is typical of the "swimlane" based process diagrams that are typically used to capture processes for BPM, usually using a notation such as BPMN (business process modeling notation), although a BPMN diagram would not normally show more lanes for the human users of the system and would not directly show the interactions with a service bus. Indeed, Figure 5 is probably already at a more detailed level of granularity than a high level business process should be documented. That pure representation of a business process, completely abstracted from the detail of interactions with back end systems, is an essential part of what makes BPM so powerful as a way of documenting, analyzing, and even implementing business requirements in a rapid and agile way, and is in itself of huge value. However, you must remember just how much this abstraction is hiding when it come to integration. The business process diagram is just the tip of the iceberg when it comes to integration. Clearly, there are circumstances when integration issues might be less of an issue; for example, when most of the business process is human-oriented rather than system-oriented, or if there is a mature SOA on which to build BPM processes, many interactions should be simpler to implement. However, this is not the general case. There are usually many new and complex interfaces -- or different uses of existing ones -- that need detailed rigor and understanding, and you must surface these as early as possible, using interface characteristics to ensure you remove the risk from the overall BPM initiative.

**Iterative interface analysis**

Because it is impractical (and impossible) to capture all of the integration characteristics in the early stages of a project, let’s look at how this process can be explored iteratively to ensure that just enough information is captured to inform the project and the design at each phase.
Capture of interface characteristics must be aligned with project phases or iterations. There are many different methodologies for defining projects and each use different terminology, but for the sake of simplicity, let's say that regardless of methodology there is always some form of two basic, well-separated exercises:

- **Solutioning**, when you discover in which systems the data can be found, and what types of technical interface are available to those systems.
- **Design**, when you look at the shape and size of the individual data operations that will be required by the solution.

On a traditional waterfall project you would solution the whole landscape based on the project requirements before moving to the next phase of design. In a more iterative or agile methodology, you would solution a relevant slice (often described as a story) of the overall problem, then move straight into design and implementation of that isolated slice of the overall picture. Either way, the two phases exist, and so we can discuss here what you should capture at each stage.

Whilst a structured approach is preferred for collecting the minimum characteristics that should be captured at each stage, nothing can replace the eye of an experienced integration specialist, who will be able to infer from the early characteristics captured that deeper investigation into some interfaces will be needed sooner than is suggested here. In the sections that follow, suggestions are included (where possible) regarding what characteristics can be useful ahead of time if they are easily available.

### Solutioning

In this phase, you are only aware of the fundamental systems involved in the architecture and the data they will provide to the solution. Early in this phase you will have nothing more than box diagrams on a whiteboard, but by the end you could have the beginnings of an **interface catalogue**.

#### System context diagram

At this point in the project, you should have created a system context diagram at the very least (Figure 6).
Figure 6. Basic system context diagram

Customer relationship management
Shipping
Printing
Billing system
Product catalogue
Order management
Credit checking
Process order

Figure 7. System context diagram embellished with basic characteristics

Figure 7 embellishes the context diagram with the available technical interfacing mechanisms (transport, protocol, data format) and principle data objects used by the process to show which system they reside in.

If there are many systems involved, the context diagram might become too cluttered, in which case you might need to capture the characteristics in a separate list. You will eventually need it in list form anyway as you capture more characteristics; this list is generally known as an interface catalogue.

Be aware that there could be more than one option for how to interact with a system. You should list them all rather than favor one at this stage (Figure 8). Until you capture the next level of detail, you cannot know which interfaces are appropriate for your use. Indeed, you might use more than one option in a single process. For example, you might do lookups (read) using web services, but use JMS (with assured delivery) for writes.
You would also normally ask about availability at this stage. For example, if a system is down for two hours at night, then a queue might need to be put in front of it to store requests when it is down, rather than perfect a direct JDBC write.

**Figure 8. Multiple ways of interfacing with a single system**

What do these basic characteristics you have captured tell you, and what more do you need to know?

- **The art of the possible:** You know whether there is an interface with each of the key systems or not. Early identification of systems that have no (or unsuitable) interfaces is a key part of the solutioning process, representing high-risk integration. However, if you have identified characteristics for your interfaces, keep in mind that you could later find that the interface doesn’t have all the characteristics that you need, so there is still risk that you might have to build a new interface, or spend time enhancing an existing one.

- **Early research requirements:** You know the core interface technologies you will need to use and thereby what skill sets the project team will need. If any of these technologies are a significant unknown, you can use this early warning to initiate research to improve your knowledge, which will help you in the next phase of the project.

- **Data strategy:** You know which systems your primary data lives in, but, more importantly, you know if it is present in more than one system. If so, you should ensure you understand if there is already a strategy in place to keep data in synchronization, or whether you will need to put that in place. Put another way, you need to identify which system is the master for each data item (the single version of the truth), and if there is more than one, how conflicting updates will be handled.

In short, you have improved your understanding of the complexity and reduced the risk -- but certainly not eliminated it. The risk could still be high, but you at least know where it lies. You must move a level deeper if you are to have any confidence in your estimates at this stage.

**Gap analysis for risk assessment and estimating**

At the solutioning stage, you will still likely need to provide an assessment of the risk involved in the project and some high level estimates. The information you have so far is simply not enough to enable you to do that.

**Table 2. Interface characteristics during solutioning**

<table>
<thead>
<tr>
<th>DATA</th>
<th>INTEGRITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal data objects</td>
<td>Validation</td>
</tr>
<tr>
<td>Operation/function</td>
<td>Transactionality</td>
</tr>
<tr>
<td>Required</td>
<td>Optional</td>
</tr>
</tbody>
</table>
The initial characteristics you captured on the solution context diagram (shown as Required in Table 2) refer only to the mechanisms by which you can exchange data with the systems. You are not yet at the level of the individual functions or operations that can be performed through that interface. For example, you might only have an understanding of the overall volumes of data that will be passed over an interface, but not the request rates for each specific data exchange. Even with this, you would be able to establish if the interface will be completely overwhelmed by the new requirements. You will reach the full level of detail in the next phase, but there are still many early warning signals you could get if an experienced integration specialist were to consider (even at a high level) a slightly deeper set of characteristics. A suggested set of further characteristics to look into are shown as Optional in Table 2.

You might be wondering where the detail of the overall volumes of data to be passed over the interface came from. This required not only knowledge of the interface exposed by the interface, but also an understanding of the requirements of the requester. At this stage, you need to do more than simply establish what interfaces the systems make available. As you saw back in Figure 2, you also need to capture the requirements of those that will make requests on those systems.

From here on, your capture is all about comparing the differences between requirements of the requestor and the reality offered by the providers. These differences will enable you to understand the integration complexities you will have to overcome. For example, if you know that the requester will require real-time access to data, but the only interface available is batch-based, you already know that interface will probably not be adequate. If there is no other interface currently available, you will have to design and implement a completely new one. You should assume this integration problem will be high risk and complex. On the other hand, you might be fortunate that there is already a fully governed SOA service available for re-use that you can use. In theory, this should significantly lower the risk and complexity of the integration although, as noted earlier, you need to consider how mature and well governed the SOA is before making any assumptions.

Finally, along with the additional characteristics noted above, you might also want to get a rough understanding of the granularity of the interfaces compared to what is required. This is difficult,
as you will not gain a detailed picture of this until the next phase. However, composition adds significant complexity to the integration, so it is a critical factor.

Since you cannot yet build an estimate on these additional factors in detail, the typical approach is to delve deeply into a representative sample set of interactions and extrapolate from there. These also provide opportunity for risk mitigation, as they can be used for proof of concept exercises.

Design

**Individual interaction design**

In the previous phase, you had no time to establish the specifics of each system interaction, except for perhaps the representative examples created for estimating.

In this phase of the project, you will be realizing the specific functional interactions of the solution by building out an understanding of the specific data exchanges that will take place across the interfaces you already defined in the design phase. These exchanges are typically represented in some form of sequence interaction diagram, such as that shown in Figure 9.

**Figure 9. Sequence interaction diagram showing specific method signatures and composition in the integration layer**

The sequence diagrams force you to establish whether the systems’ interfaces provide the specific data operations that you require. This implicitly populates the remaining characteristics in the Data section. It is important that you use the same gap analysis technique you used earlier, understanding the problem from the requester’s point of view, and then consider what the provider has to offer.

**Interface granularity and composition**

There is some key information that will spill out from the creation of sequence diagrams – whether or not the granularity of the available interfaces is correct. It is often the case that the requester has a more course-grained functional requirement than the provider’s interface. For example, the requester requires a customer, with any current orders, and a list of the high level details of
any accounts held. This might require several requests to a provider, and maybe even to multiple providers. Figure 9 shows a further example of a mismatch in granularity whereby order creation and shipping need to be combined into a single operation to "process the order." It’s easy to see why the sequence diagrams are so important at this stage.

This difference in granularity means you will need to perform composition (multiple requests wrapped up as one). The macro design must establish where this composition should take place: in the requester, in the integration layer, or in the back end system itself. The decision will be based on a balance between factors such as performance and re-use. (Where and how this composition should take place is beyond the scope of this article.)

What is relevant to the core concept of this article, however, is just how important the other characteristics become if composition is required. Here are some common examples:

- If a composition requires more than one interaction that changes data, you need to consider whether those interactions can be transactionally combined, and, if not, how you will perform error management if the request fails between the updates.
- If the individual requests of a composition need to be performed in parallel in order to meet the response time requirements, do you have sufficient concurrency to handle the increased number of connections required. And if the system interfaces are thread-blocking, how will you spawn and control the additional threads required?
- If a composition interacts with multiple systems, how will you retain the separate security information and any other stateful context such as sessions and connections required?

**Establishing the integration patterns**

Once you know the specific operations that will be performed, you can then explore all the other characteristics in the proper detail for each operation, both in terms of the requestor's requirements and the existing interface's current capabilities.

As you compare the characteristics of requestor and provider, you can then establish the integration patterns that will be required to resolve the differences, as you saw in Figure 3. Be aware that the "composition" we have just been discussing is itself one of the integration patterns.

At this stage, the patterns are defined at a logical level, without reference to the specific technology. However, as commonly used integration patterns are well established, standard implementation techniques within the available technologies should also be explored, tested, and documented. This will significantly streamline the micro design phase, next.

**Micro design**

In the micro design phase, you use the captured characteristics to complete the technology-specific aspects of the design.

Notice that all the interface characteristics are agnostic of the integration technologies; they are logical characteristics. Therefore, capturing interface characteristics should have been completed by the time you reach micro design, and you are purely using the characteristics to aid you in
the final details of the design. One of the most common integration-related causes of projects running over budget is ineffective capture of interface characteristics prior to micro design, leaving inestimable -- possibly unresolvable -- issues for this phase, or, worse still, for implementation. By far the most common and most troublesome issue here is failing to assess and capture a sufficient amount of the data characteristics prior to micro design.

Beware of good intentions that aim to "add that on later," especially for characteristics that are cross-cutting concerns, such as security. It is acceptable to defer something only once you've established how complex it is, and how much harder it will be to add it onto a running implementation.

The primary activities in this phase that make use of the interface characteristics are:

- Bringing precision to the data model, such as defining of detailed data types, and restructuring of data required for physical representation.
- Detailing the technology specific implementation of the integration; what technologies features will be used, and how.
- Designing for performance, using the performance characteristics to assist in choosing between multiple technology options.

Conclusion

This article described a mechanical way to analyze integration requirements using interface characteristics, and discussed why this is of such critical importance, not only to traditional enterprise application integration but to other enterprise initiatives such as SOA and BPM as well. This article also looked at how this technique can be used iteratively across the lifecycle of a project to ensure you have the right information at the right time.

Part 2 will look at each individual interface characteristic in detail to help you acquire a clear understanding of their meaning and importance.

Acknowledgements

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

- Enterprise Service Bus, re-examined
- Increase flexibility with the Service Integration Maturity Model (SIMM)
- Solution design in WebSphere Process Server: Part 1
- Patterns and Best Practices for Enterprise Integration
- Executing SOA: A Methodology for Service Modeling and Design
- Enterprise Connectivity Patterns: Implementing integration solutions with IBM's Enterprise Service Bus products
- Solution design in WebSphere Process Server and WebSphere ESB, Part 3: Process implementation types: Patterns based design for process-based solutions
- developerWorks BPM zone: Get the latest technical resources on IBM BPM solutions, including downloads, demos, articles, tutorials, events, webcasts, and more.
- IBM BPM Journal: Get the latest articles and columns on BPM solutions in this quarterly journal, also available in both Kindle and PDF versions.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)