CeTN Changengine Technical Note

Application Integration (EAI) Basics

Summary

This CeTN looks at the basics of EAI - Enterprise Application Integration.

We look at the areas of Application Integration and Data Integration, and discuss what's involved to get the application systems communicating.

We then look at how to add the process management layer to this.

At the end of this CeTN you will understand how we can build solutions with Changengine and Middleware.

Notices

Printing History

First Published September 2000 (for Changengine A.04 and later)

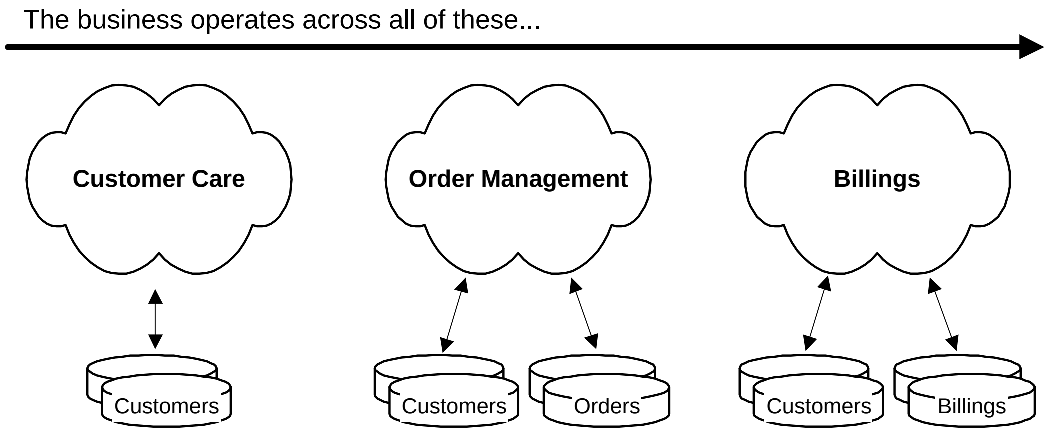
Restricted Rights Legend

Contents

Introduction

Most sites invest heavily in software packages. They may have Customer Care systems, Order Management systems, Billing systems, etc. and these may be already in place and working. All of these will have been purchased (or written) to address a particular business need or function, and they are probably all very good at what they do. However, they will each have their own view of how the business works.

We need to view the business right across all of these applications:



However, these systems tend to operate completely independently of each other. Each system typically has their own view of their data - held within their own databases and with their own data formats, syntax and semantics - and they will probably have little or no idea about interfacing to anyone else's software. It's easy to understand how this can happen - each package was probably written by a completely different software house. Indeed, even if the packages are written by the same software house there can still be issues when it comes to integrating them :-(

In the diagram above I show that the Order Management system maintains data about the actual orders placed by our customers, and it also maintains its own customer details. The Billings system holds all our billings details as well as its own customer details. And, of-course, our Customer Care system holds a bunch of details about our customer base. Each system will hold slightly different information about the customer - specific to the requirements of that package. So in this example we have 3 different definitions for “the customer”, held across 3 different data sources. I actually heard one site say that they currently had 8 different definitions for the customer. You can imagine the heartache should one of their customers ever decide to change their phone number!

I think you can start to appreciate how getting these application packages to synchronize their data and communicate with each other would be a huge step forward! ...and with an integrated set of applications you can construct powerful, effective business processes!

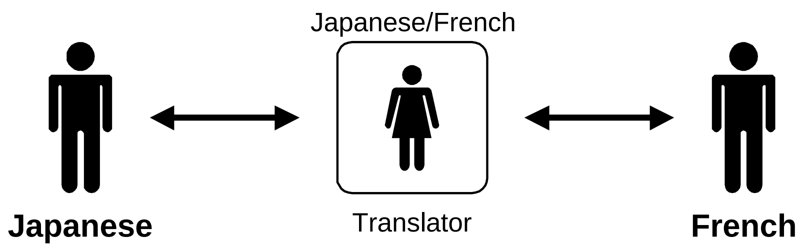
Let's first consider the whole issue of how we go about integrating our application software packages and getting them to communicate with each other...

A Human Analogy

Imagine that you have two people who need to talk to each other. One only speaks Japanese. The other only speaks French.

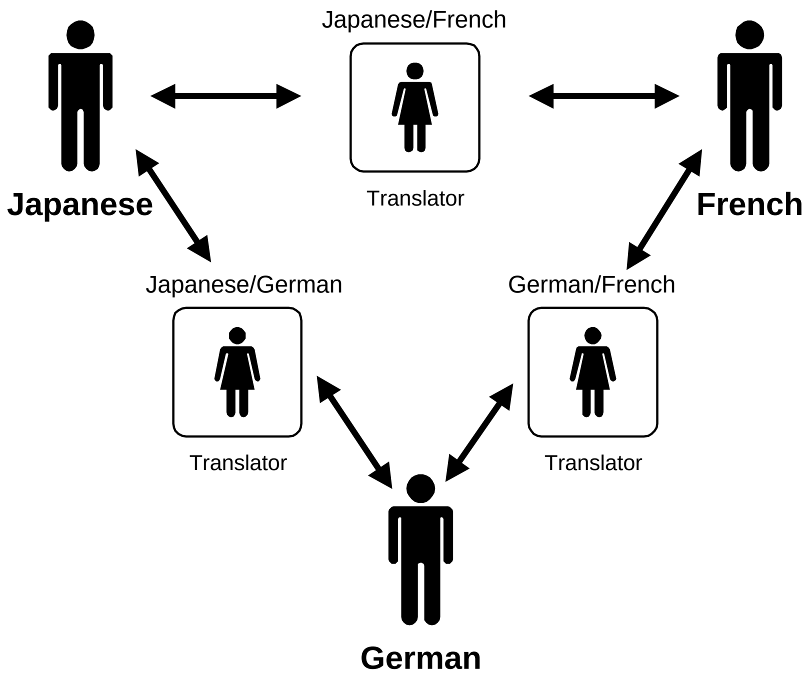
To allow them to communicate you need to find an "Interpretor" (or "Translator") - someone who can understand and speak both Japanese and French. You then put this translator between the two people and they can now carry on a conversation.

It might look something like this:



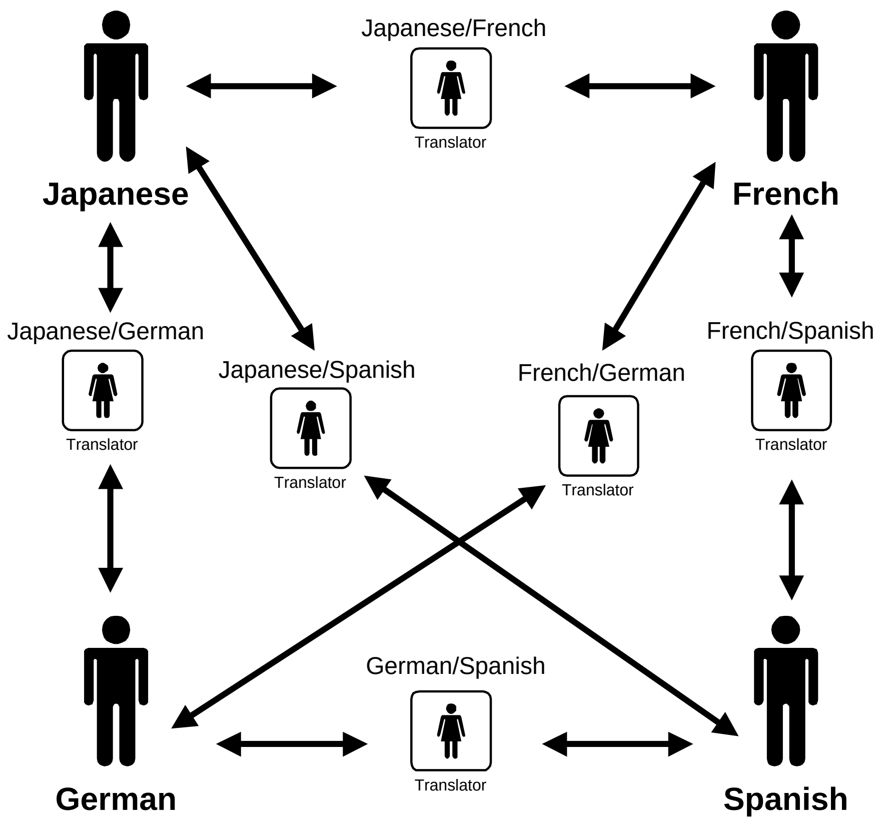
Now suppose that another person joins the group and he can only speak German. You need to allow this person to talk with both the Japanese person and the French person. Well, you need to find a German<->Japanese translator and a German<->French translator.

It now looks something like this:



Suppose a fourth person comes along, and they only speak Spanish! To allow them all to speak with each other requires a few more translators. We require 3 more: French<->Spanish, Japanese<->Spanish and German<->Spanish.

It would look like this:



I think you're starting to see that this system of providing a translator for each individual conversation is OK when you have only 2, or maybe 3, people, but it clearly becomes cumbersome when you add a 4th person - you end up with more translators in the room than you have people! Indeed, if you were to add a 5th person to this network you would require a total of 10 translators.

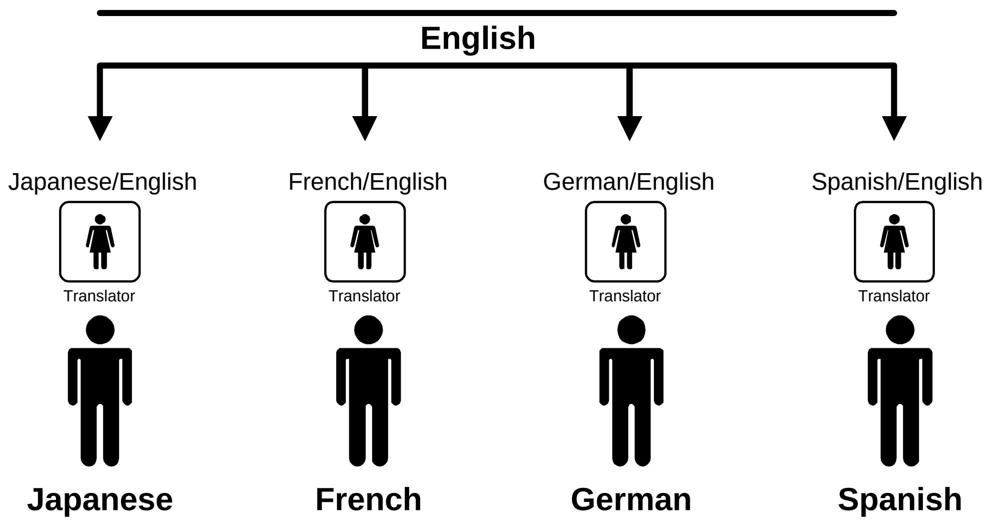
For what it's worth, the actual number of translators required for N people speaking different languages is: N(N-1)/2

There is an alternate method...

If instead of allowing every single person to have a "direct" conversation with everybody else in the room, what if you chose an independent language and declared that as our language for communication. We would then hire translators to/from that independent language?!

Let's suppose we take that room of 4 people (Japanese/French/German/Spanish) and this time decided that for our communication, English would be our independent ("Common") language.

We could now redraw the room as follows:



With English as the “Common” language we now only need one translator per person.

So, whether we have 20 people of different languages, or 5, it’s still 1 translator per person.

Also, if the Japanese speaking person should happen to leave the room and be replaced by a Dutch speaking person, we would only need to replace one translator!

So you (hopefully) see that if we can get everybody “talking the same language” things become a whole lot simpler.

Now, ok, I can hear you all saying things like: “But this means that we translate everything twice?!”, and “Surely there are some things that just don’t translate properly from Japanese to French or whatever!”, and “Doesn’t it get worse if we then have to translate this again?”...

And these are all valid comments...but it is just an analogy, and it is one that helped me understand the basics of EAI...and that’s where I want to move on to.

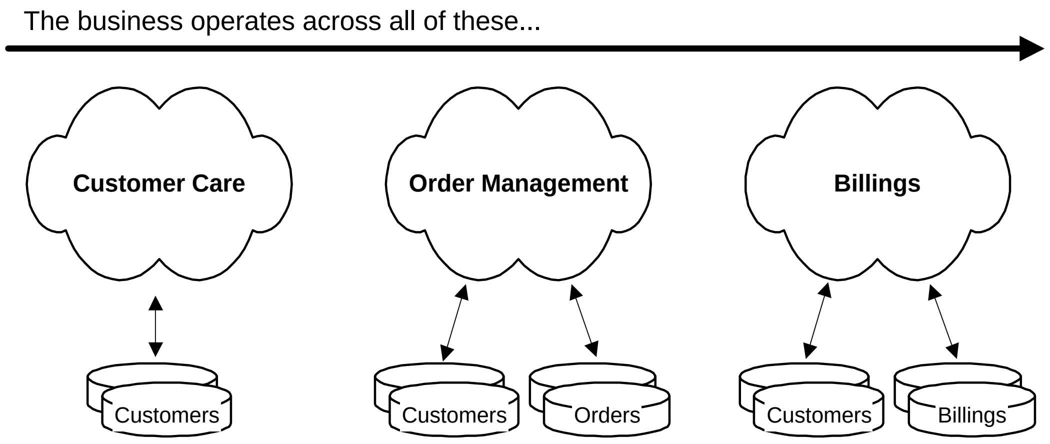
So, at this point, all I am trying to show you is that if we can get everybody “talking the same language” then things can become a whole lot simpler!

And this is a key part of what EAI technology is all about!

Application Integration

As we said in the introduction to this CeTN, your business runs a number of software packages. These packages each carry out certain aspects of your overall business. Some might be old systems that you may one-day replace. Some might be fairly new. Some are excellent, and some are not so good. But for the moment, you have these systems and together they help you run your business. However, there is little or no communication between them. They each work as individual packages - doing what they do and that's that. And, as we said, your business works across the lot of them.

Using our earlier example:



Let's just consider this example and think through how you could enable some of these packages to communicate and be more integrated with each other:

- To enable the Customer Care system to see if orders have been placed by any new customers:

  You could write a batch job that runs (say) every night and have it interrogate the customer database of the Order Management system. When it finds a new customer it could enter these details into the customer database within the Customer Care system.

  To do this requires knowledge of both the customer database structure within the Order Management system and the customer database structure within the Customer Care system

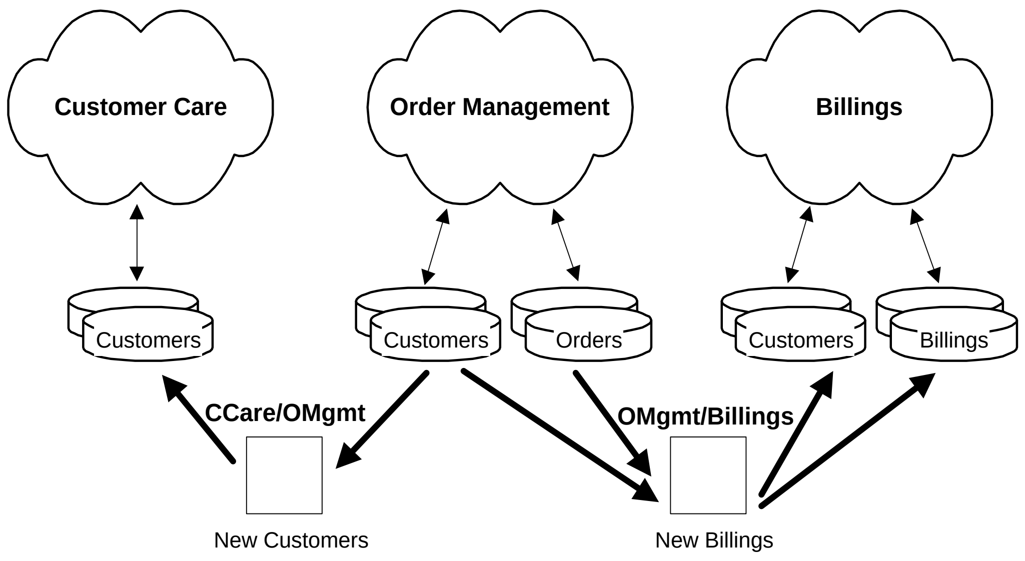
- To enable the Order Management system to automatically kick off a new billing once the order has been fulfilled:

  You could write a batch job that scans the orders database tables to see any completed orders for that day. It could then look up the customer information from the orders customer database and with all the necessary information it could make an entry into the appropriate database tables within the Billings system. This of-course assumes that the billing system is capable of detecting these new entries and therefore kicking-off these billings.

  Or maybe the Billings system provides a "file import" system whereby, instead of populating its database tables, you produce a file in a special format for it to read. Maybe it then has a batch import function that can be automatically run. Maybe, instead, this batch import has to be run manually by some operator the next day. It all depends on what the Billings system supports.

  To do any of these options requires knowledge of how these two systems (Orders and Billings) work internally.

We can represent this diagramatically as follows:



Interestingly enough, we have effectively written two “translators”:

- One to translate “New Customers” from the Order Management system to the Customer Care system ...and...
- One to translate “New Billings” from the Order Management system to the Billings system

And just like language translators, each of these translators requires intimate knowledge of each system - such as database structures, table names, data item names/types, etc.

Clearly, these two translators we have written are just the beginning. Over time we would determine new features of integration that we would like to offer between each of these systems and add these to our translators. For example, to the CCare/OMgmt translator, we would probably add the ability to detect not just new customers but updated customer information, and have this information sent across. Indeed, we would probably add the ability for changes to the Customer Care database to be sent over to update the Order Management customer database, and hence make it a 2-way translator. And we’d no-doubt add more abilities to the OMgmt/Billings translator and eventually make that a 2-way translator.

But you can probably see what’s coming can’t you?

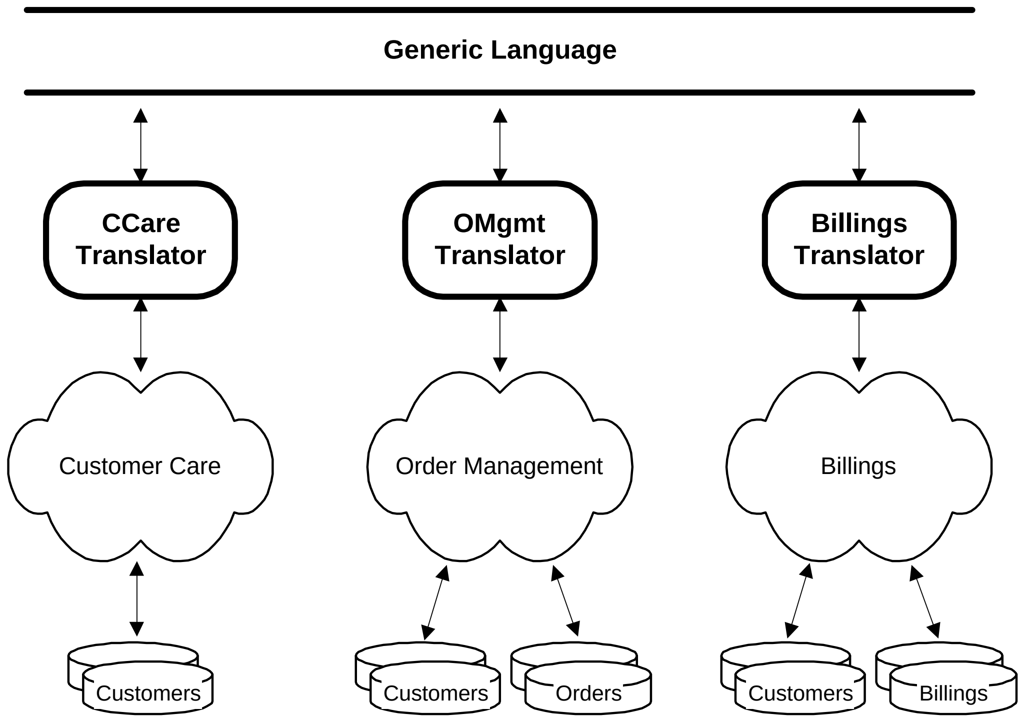
If we were now to go about integrating the Customer Care system with the Billings system, we would need to write a third translator. What then happens if/when your company buy-in a fourth application package? You have to write three new translators!!! One to/from the Customer Care system, one to/from the Order Management system, and one to/from the Billings system. That’s a lot of work!

And what happens if (say) a year later your company decide to upgrade the Order Management system from Version 1.0 to Version 2.0? and this new version uses different database structures? All you translators (the CCare/OMgmt, the OMgmt/Billings, and any others) have to be re-written to understand these new layouts! That’s a massive task!

What if we agreed on a “common” language?

Instead of writing all these individual translators between each and every system - coding in the specifics of each system into every bit of code we write - what if we agreed to provide translators to a “common” (or “generic”) language?

It would look like this:



Let’s consider how we might “integrate” things with this new setup.

Earlier, we discussed a way to enable the Customer Care system to see if orders have been placed by any new customers. Let’s see how this might be implemented under this new “Generic Language” scheme:

- In the “OMgmt Translator” we would write a routine that did all the digging around within the Order Management data tables to retrieve any new customers - we might call this routine: OMgmt::GetNewCust()
- In the “CCare Translator” we would write code that makes a call to this OMgmt::GetNewCust() routine and then takes the output and writes it into the Customer Care database.

With this system we see that all the intelligence and knowledge about the Order Management system is now totally within the OMgmt Translator, and all the specific knowledge about the Customer Care system is coded totally within the CCare Translator. Thus all product specific knowledge and access is coded within that product’s translator. (This is straightforward object oriented design principles!)

This also means that if we were ever to upgrade the Order Management system from Version 1.0 to Version 2.0, we would only need to rewrite the OMgmt Translator! So long as we made the same routines available - such as the OMgmt::GetNewCust() routine - we’d be able to slot in Version 2.0 of the Order Management system without affecting any of the other systems.

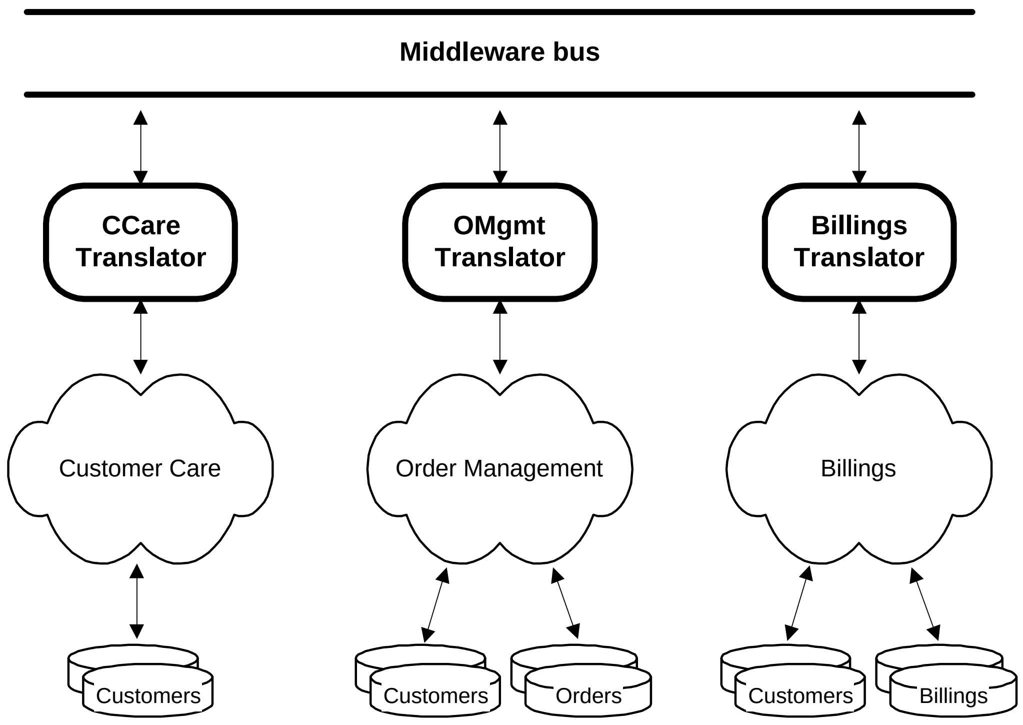
But how does the CCare Translator “make a call” to this OMgmt::GetNewCust() routine?

As the above diagram shows, the CCare translator and the OMgmt translator are indeed quite separate pieces of code. For one translator to be able to call routines in another we clearly need some sort of “middle man” to pass these messages around. And indeed, when you use this method for integrating your applications you buy in this “middle man” - it’s called **Middleware**.

The Middleware you buy will have its own “Generic Language” and it will provide a developer’s kit and API to allow you to write translators that can plug-in to the Middleware bus and make calls to other translators on the Bus.

The company who wrote the Middleware will also be able to sell you pre-written translators for many existing applications (such as SAP, Clarify, Portal, etc.). So it might be that you can buy-in the necessary translators for your applications and not have to write any real code yourself!

To reprint the earlier diagram but show that it is a Middleware bus:

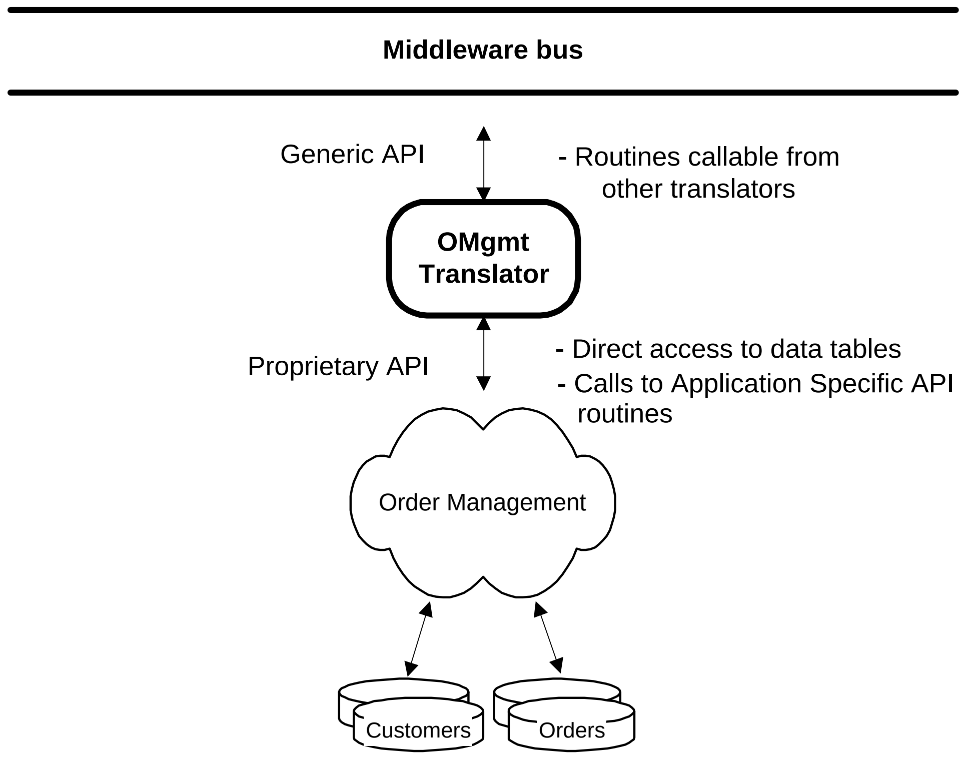


So we see that by using a Middleware bus (it’s often called a “Middleware Message Bus” because it allows messages to be passed between translators), the idea is to write translators that provide a set of routines that you can call from other translators. For example, we talked earlier about how the Order Management translator could provide a routine called OMgmt::GetNewCust(). All the nitty-gritty of exactly how this translates to accessing files and data tables within the Order Management system is hidden from the outside world. The translator makes it look as though the Order Management system has a generic, callable API, in the language of the Middleware bus.

When we talked about how we might write the routine OMgmt::GetNewCust(), we said that it would probably access the orders database directly and then the customer database directly and then return the results. In other words, we talked about it simply attacking the internal data structures of the Order Management system directly. Some application systems come with their own API. It may be that our Order Management system does provide routines that you can call to locate new customers. Whether our application system provides its own API or not, we still need to write a translator that will make use of the application system’s API if it is there, and then map (translate) these to a generic set of routines that are callable across the Middleware message bus.

In other words, the translator maps from any proprietary API (if there is one) to a generic API.

We can represent it as follows:


Middleware bus
Generic API
- Routines callable from other translators
OMgmt Translator
Proprietary API
- Direct access to data tables
- Calls to Application Specific API routines
Order Management
Customers
Orders

I mentioned earlier that you could buy translators from the Middleware suppliers.

They obviously want everyone to buy their particular flavour of Middleware, and so they will have pre-written a bunch of translators for many of the standard software packages. Typically there are translators available for applications such as: PeopleSoft, Clarify, various SAP versions, Portal, etc...

When they write these translators they do not simply hard-code in a set of functions that they make available. You don't just buy the (say) SAP R/3 translator and have it come pre-configured with "GetNewCust()" and "UpdateCust()" calls. No no no... It's quite clever really. When you buy a pre-written translator it usually comes with a configuration tool whereby you can program it with the functions you want and tell it which data sources within your application system it should access.

In other words, pre-written translators are configurable **-** in the EAI world the term they typically use for this is to say that the translator is **Dynamic**.

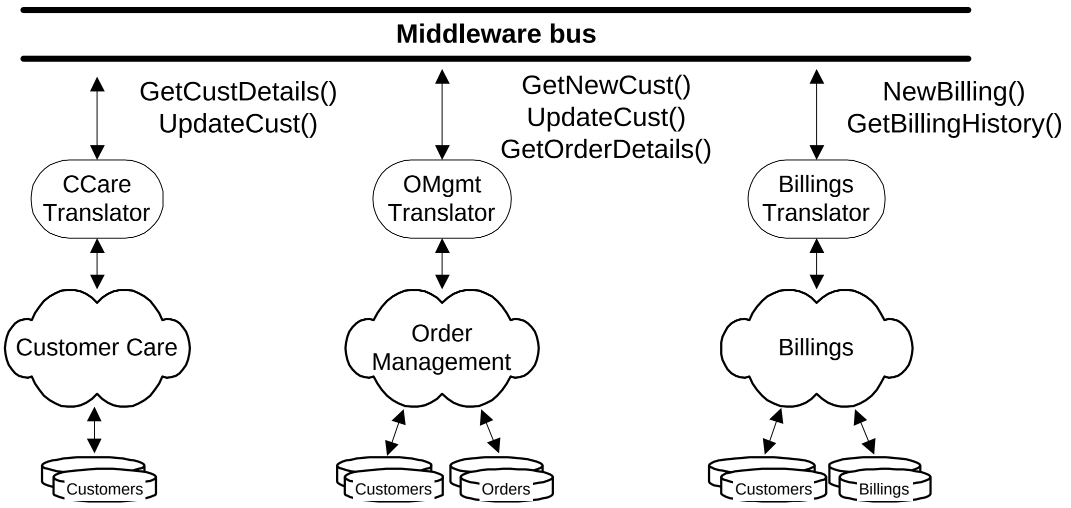
With a dynamic translator you can configure the functions that you wish to make available and exactly how each of these retrieves its data from your application system.

So, EAI means that you end up exposing a set of function calls (generic API) for each application system.

The application system might have had its own API, in which case it is reasonably easy to build a generic API callable from the Middleware bus. Of-course, some systems may not have had much (if any) of an API. It may be that the system was originally written to be accessed from PC screens and users running GUIs and not meant to be accessed electronically from other applications at all. But that's the whole point of EAI - for an application system to be callable from another, someone has to spend the time analyzing the software and determining exactly how to create an API for it. Like my little OMgmt::GetNewCust() example, that was done by accessing the databases directly - something the standard application interface never allowed. So these translators are very specific to the application software (and version) for which they are written, but their job is to provide this generic API - callable from the Middleware bus.

Whether the application system had its own API or not, you have now constructed a callable set of routines that mean the application system can now be called via software.

So if we show our example, listing some of the calls that might be available, we might get something that looked like this:



In this case we see that the OMgmt Translator has three calls available. You can ask it to GetNewCust() - and it will retrieve any new customers that have placed orders since the date you supply in the call. You can also ask it to UpdateCust() where it takes the details you give it and it updates its customer records. And you can ask it to GetOrderDetails() where it will retrieve the details of all the orders for the given customer.

So too, the Billings system can be asked to start a new billing (NewBilling()) - this would be called from the Order Management system when it has fulfilled an order. You can also ask the Billing system to GetBillingHistory() and retrieve all the billing records for the given customer.

And so it goes...

With EAI technology I can now connect up my applications and they can "talk"!! For each application I can "expose" certain functionality to the rest of the applications. For example, the Billings system now exposes the fact that it can be told about a fulfilled order and it can go ahead and bill that customer. It has done this by having the NewBilling() call available on the Middleware bus.

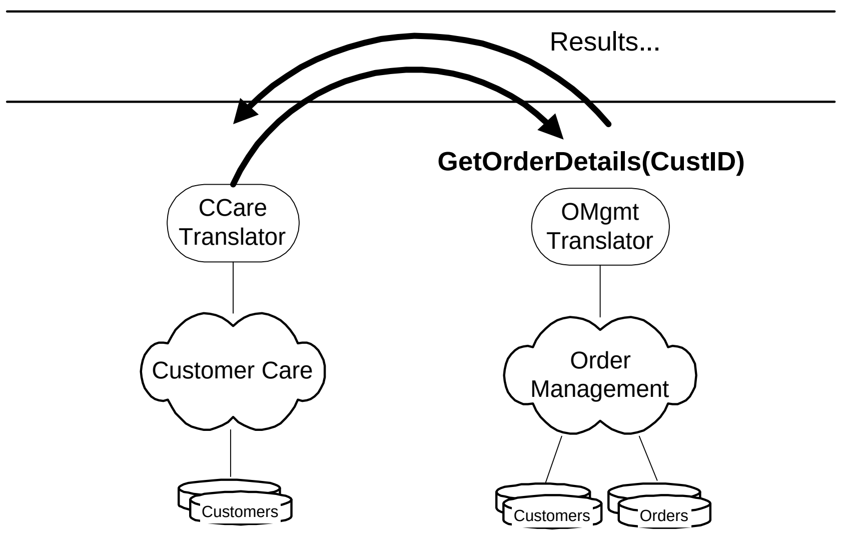
My application systems are now "talking the same language"!

Data Integration

This is a point that is often overlooked when discussing how to integrate your applications.

Hopefully by now you understand the basic idea of EAI and you can visualize setting up your applications to call each other and pass around information. But this information that’s being passed around...is it all in the same format?

Earlier, when I showed an example of what functions the translators might support, I showed that the Order Management translator might support a function called “GetOrderDetails()” where you passed it the Customer ID of the customer you were interested in and it would return certain details about the orders they had placed. It looks something like this:



What’s to say that the Customer ID in the Customer Care system is the same as the Customer ID in the Order Management system? Indeed, I’d be most surprised if they were the same.

It’s quite possible that within the Customer Care system, the Customer ID is a simple number - they started with Customer ID 1000 and each new customer has been assigned numbers incremented from there...1000, 1001, 1002, etc...

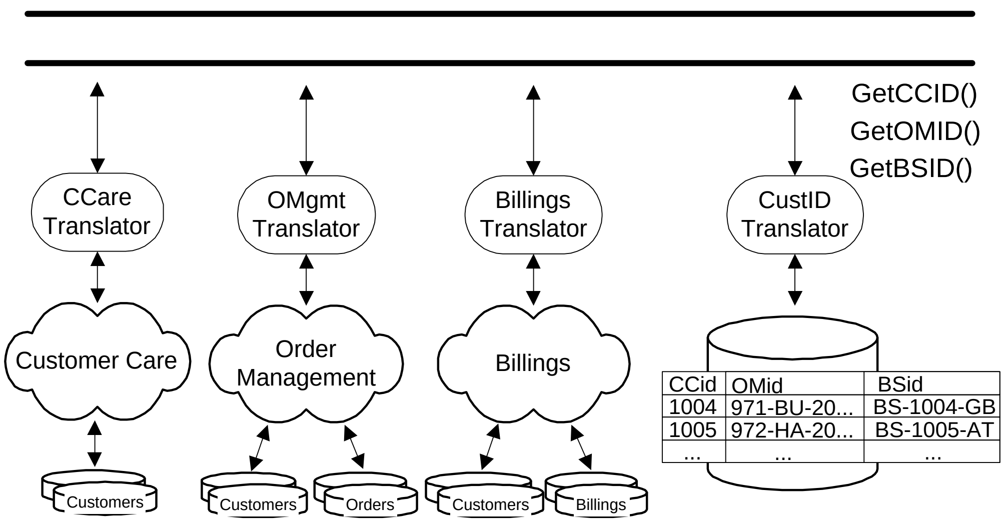
But in the Order Management system the Customer ID might be a string made up of a unique number, plus a code to identify the location of their offices (for internal support reasons), plus the date their entry was first created - something like: 573-BE-19990507. (Ok, it’s a horrible looking number, but I’ve worked on sites that have had much worse :-)

Because this Customer ID field is being passed around between systems we need to set up some way for cross-referencing between these systems. We need some sort of mapping that says that Customer ID 1004 in the Customer Care system maps to Customer ID 971-BU-20000102 in the Order Management system.

One way to handle this is to build ourselves a separate database that will contain all the Customer IDs from either system listing them in pairs. But what about the Billings system? We need to check its definition for Customer ID and include that. So our database needs to contain entries showing the three possible Customer IDs for each actual customer.

Because we are going to be asking for information keyed on Customer ID we must map this between all three systems.

It might look like this:



CCid	OMid	BSid
1004	971-BU-20...	BS-1004-GB
1005	972-HA-20...	BS-1005-AT
...	...	...

Where:

- We have gone through the three systems and matched-up the three versions of the Customer ID and written these to an external database.
- We then need to provide a translator so that this data is visible on the Middleware bus.

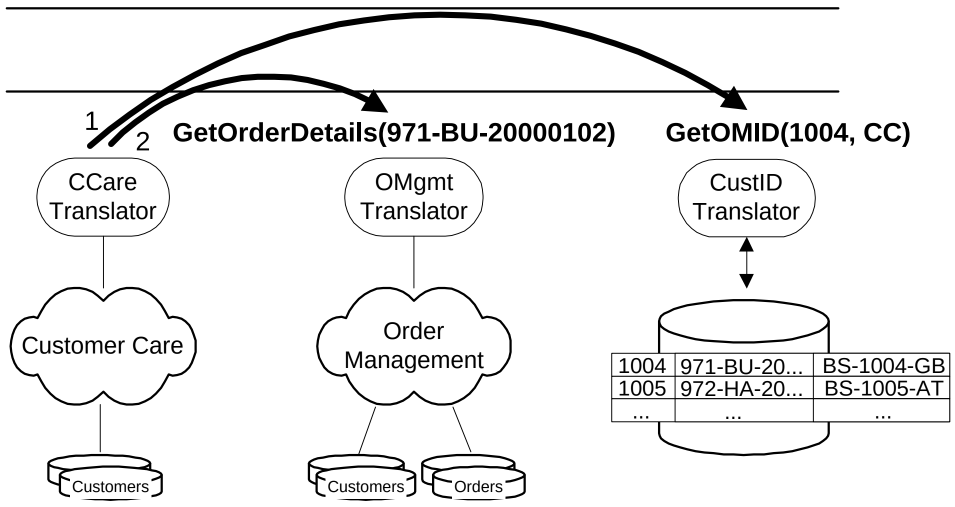
  This translator will be configured to accept three calls, where you pass in the Customer ID that you have and what type it is (CC, OM, or BS). It then returns the equivalent ID.

This is quite a common task when integrating applications, and most Middleware suppliers do provide database translators for the major databases available today. So, configuring a database translator to offer these three calls is very simple.

The real issue is the time it takes for you to construct the data by going through the existing application systems.

Now if we go back to our example where the Customer Care system was asking the Order Management system for “GetOrderDetails(CustID)”, we see now that the Customer Care system has to make two calls to achieve this.

It looks like this:



Where:

- It first asks the CustID translator to translate the Customer ID from “1004” to its equivalent in the Order Management system.
- It then uses the returned value (971-BU-20000102) to ask the Order Management system to supply the order details for this customer.

This issue of looking through the application systems that are to be integrated and seeing what data mappings and conversions are required is a major part of the overall integration effort.

You need to look at all the data models of all the applications being integrated and decide what needs to happen and how you will resolve it. Issues like the fact that a data item is called ABC in one system and it’s called DEF in another isn’t a problem - simple name mapping is usually handled in the Middleware. So too, basic data-type conversions tend to be handled within the Middleware.

And of-course, there’s the question of data synchronization. In our example we have three application systems (Order Management, Customer Care, Billings) each holding their own definition (and data) of the customer. So does it matter to us that one might be updated without the others seeing that update? You might think the answer is an obvious “Yes”, but for some companies it may not be. Having said that, it is very common to see people set up triggers on each application system such that when the customer record (or whatever) is updated in one system, this update is sent across to the other systems so they can incorporate the changes.

All these issues of deciding how to synchronize your data, choosing which sources are your masters and which are your slaves, setting up any addition databases/tables for mapping key fields (such as I showed for the various definitions of Customer ID), etc. - these are all part of what we call **Data Normalization**.

Once you have a fully normalized data model you have effectively agreed a meta data model across all of your applications.

As you can probably now appreciate, data integration and normalization is a major part or the overall EAI effort, and one that is not to be underestimated!

Summary

- We’ve seen that to integrate your application systems using EAI technology, involves the use of a central, generic message bus (Middleware) across which all the applications will talk - via translators.
- You then write (or buy) translators (often called “Adapters”) that allow your application systems to plug in to the Middleware message bus, and thus send and receive messages.
- You only need one translator per application system.

  You can quite happily have more if you want - so they can share the load - but one is enough to get you going.
- A translator makes the application system look like it has a generic, callable API in the language of the Middleware bus.
- A translator hides all application specific details (file structures, internal API calls, etc.) from the Middleware bus, and thus from the other application systems.
- Using a translator gives you “plug-and-play” capability.

  You can upgrade an application system from Version X to Version Y without the other systems knowing about it. All you need to do is to rewrite the translator to access the new version of the application system, and make sure that you keep the same set of generic API calls.
- You can buy translators for many of the major application systems.

  These translators are typically *Dynamic* in that they can be configured on your site to say what functions they expose and how those functions are implemented.
- You will need to *Normalize* your data.

  This will mean going through and resolving issues of data duplication, synchronization, and semantics. Data Normalization is a major task, and one that you should not underestimate.

Great! So that’s it! Off you go and expose all sorts of functionality for all your application systems ...but... what functionality should you expose? Who says that a GetBillingHistory() function is a worthwhile thing to make available from your Billing system? I mean, let’s face it, these application systems never used to talk to each other, so who is suddenly going to want to make this GetBillingHistory() call?

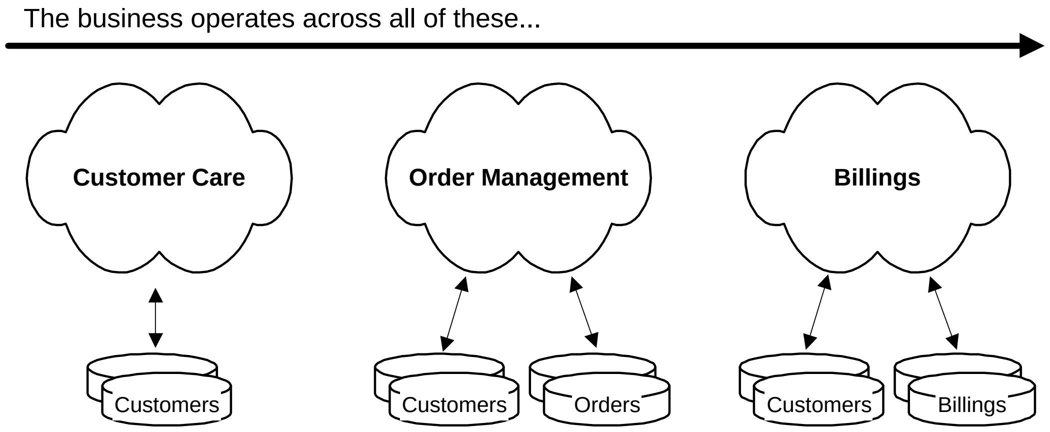
It’s at this point that most people realize that to carry out application integration as a stand-alone project is not the right thing to be doing. Buying-in the necessary Middleware to enable your application systems to talk is essential, but the decisions as to what functions need to be exposed from each application system needs to be **driven by your business processes!**

Once you are comfortable that you have the right Middleware and the right translators, you should then step back and look at the business processes that you need to define - the processes that are actually needed to run your business - the processes that span these application systems. By defining these business processes you are then in the perfect place to be able to determine exactly what functions need to be made available from the individual application systems.

It is the business process that drives the application integration!

Adding Process

Remember the original diagram?



We have various systems, all carrying out their particular part of the business. We now need to operate across all of them. We need to build processes that span these application systems and make them appear as one homogeneous system.

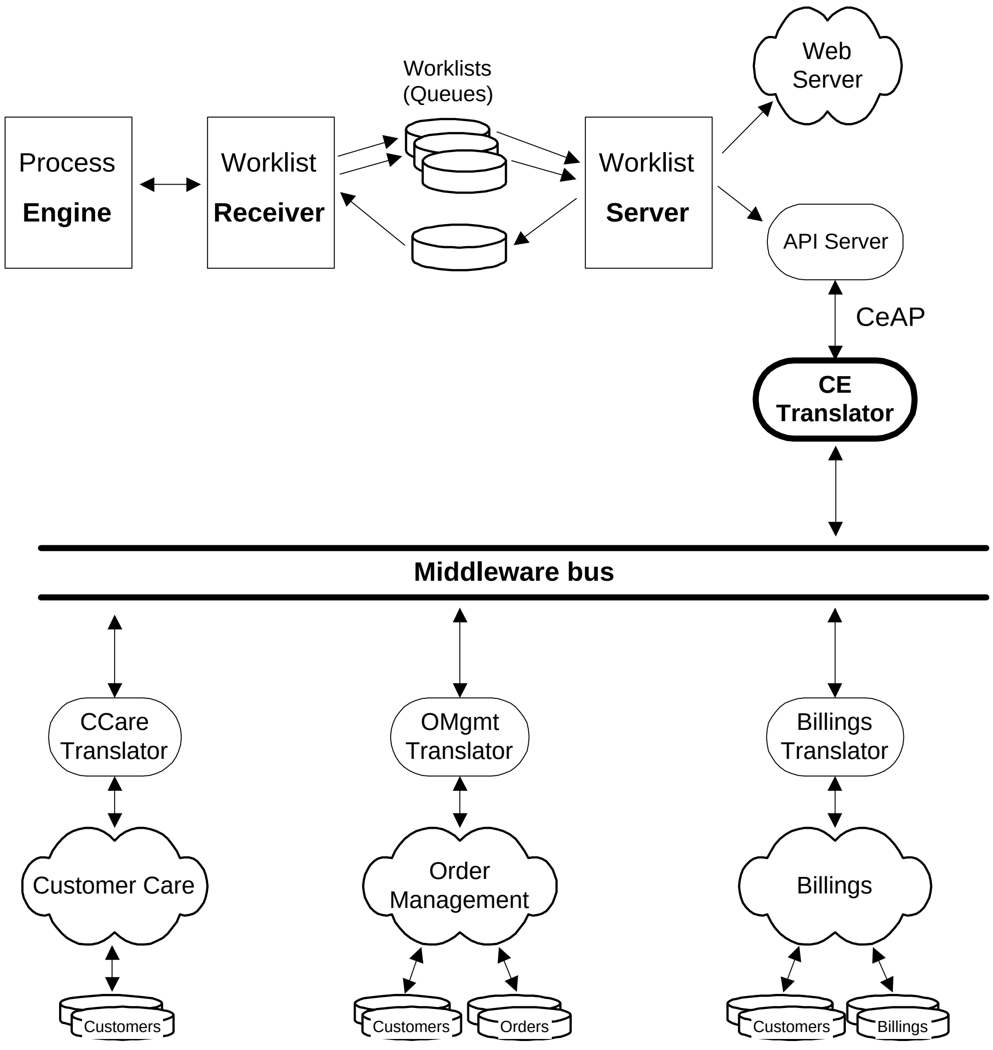
We now understand that EAI technology allows us to look carefully at these underlying application systems and decide what functions and services they currently provide to the business, and then expose these as callable functions.

All we need now is a translator for Changengine so that it can plug-in to the Middleware bus and call these functions. Indeed, we don't just want a one-way conversation - we also want to allow the application systems to call Changengine. You can easily imagine a situation where you would like (say) the Order Management system to call Changengine whenever a new order is placed and have it kick-off a process to manage the fulfillment and payment of that order.

Architecture

To write a translator for Changengine that allows it to plug-in to the Middleware bus, the translator simply needs to use the CeAPI to access a work list. The translator then looks to Changengine like any other Changengine client or user. It logs on to its work list and from there it can start processes, and handle any work items that are sent to it.

It looks something like this:



The CE Translator (usually called a "CE Adapter") simply connects to the Middleware Bus and "talks" the same language as all the other translators.

On the Changengine side of things, the CE Translator (adapter) logs on to its work list just like any normal Changengine client or adapter. It uses the CeAPI to connect to the API Server (port 9123 by default), and accesses its personal work list via the Worklist Server. It could also connect-in via the Web Server (port 80 by default) and work just fine...but using the API Server is faster!

Defining the Process

It is the process that drives the application system integration.

Once we have analyzed and designed our process, and determined what services we require from the underlying application systems we can then go about the application integration knowing exactly what functions and abilities we need to expose to the Middleware bus.

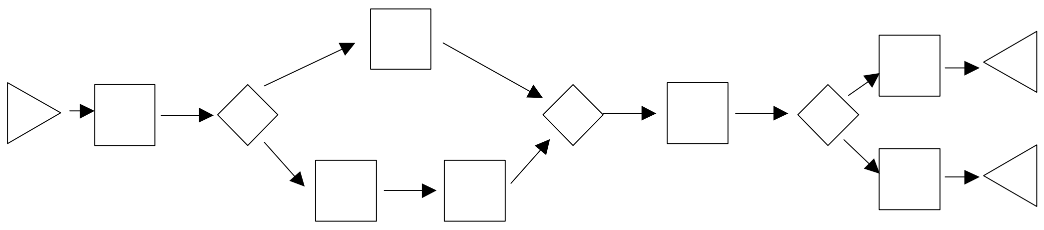
So when designing your business process, you should assume that you have a fully integrated set of application systems at your disposal - you probably don't as yet, but assume that you do have all these application systems integrated. You then design the process and as a result of that you end up with a set of services (functions) that you need from the underlying application systems. You can give this to the EAI team and they then have a specification of the functions they need to expose from the underlying application systems.

Now they might find that when they look into it, there are some services (functions) that they can not offer, in which case you need to work with them to determine a compromise. You would need to make adjustments to the process definition to compensate and design a different service call. It's an iterative situation, but because the starting point is the overall business process, you all have a clear understanding of exactly what the application integration effort must achieve.

Let's now consider how we might design a process, and the basic steps we would go through...

As with all business processes, you need to get the right people involved, go through and determine all the steps that have to happen, determine which can happen in parallel, which are sequential, where the loops are, etc. etc..

So let's say you design your process and it looks something like this:



Typically, when initially prototyping this process, we define all the data that needs to be fed in to the start node. We might define a whole host of data items, such as:

```
Cust ID
Cust Name
Cust Address
Cust Postcode
Cust Phone Number
Order Number
Order Details
Billing Address
Purchase Order Number
etc...
```

This has the convenience that we can then easily demonstrate the process and see if it meets our requirement. However there are some important areas that need to be considered before we would look at going live with this process:

- This process is carrying around with it all the data.

We definitely do not want to duplicate application data here within the process. Instead we want the process to refer to the real data that is maintained within our application systems. Where ever possible, keep the application data out of the process model.

- This process assumes that all these steps happen just for this particular business process.

  We need to consider which steps, or sequences of steps, are potentially reusable within future process within our business. Reusability is a key point to build business processes.

- The process may well be started by one of your external application systems.

  It is highly likely that the process is started by one of the external applications (Order Management, Customer Care, whatever) and not by some person logging onto Changengine and typing in data.

Let's look at these in more detail...

Application Data within the Process

We've just spent the first half of this CeTN understanding that we probably have data duplicated across potentially many existing application systems (Customer Care, Order Management, etc...) We certainly do not want to implement another system that duplicates data!

Rather than the process carrying around a copy of all the data, you want to try and aim to just carry around the various ID's for the data objects involved. Let the underlying application systems maintain the application data, you just need the process to know an ID for the customer, an ID for the particular order, etc..

Indeed, more than that, when you are first defining your process don't even concern yourself with where the Customer ID (or whatever) is defined in your existing application systems. Remember, your application systems might all define it differently (1004 in the Customer Care system, 971-BU-20000102 in the Order Management system, etc.). That doesn't bother you when defining your process. All you need to worry about is that you need to have a reference to that customer ID (or whatever) within your process.

Go ahead and define your process assuming a normalized data model. Once the process is defined, you will then see exactly which data items you need and can then investigate how best to supply these via the Middleware. In other words, let the application systems maintain the application data and have the EAI effort handle all the normalization and synchronization issues.

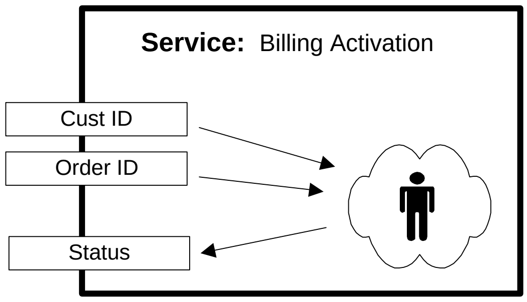
Let Changengine handle the sequencing of the work that needs to be done.

Functional Decomposition of the Process - Reusability

When defining the business process, **reusability** is a key point! You need to break it down into its constituent services and be constantly looking at what services and sections of your process might be reusable for building other processes within your business.

A Changengine service defines the interface between the business process and the resource (person or application) that will actually perform this activity.

For example:



We then connect these services into our business process via work nodes. The work node will basically specify which Changengine service it calls at that point in the process - passing the appropriate process data.

Don’t forget that a Changengine service is reusable. That is, we can define a service that needs to happen (it might be “Billing Activation” (as shown above), “Raise a purchase order”, “Run a credit check”) and then define it as a globally reusable service within Changengine.

Also, by identifying a service as a potential for reuse in later process definitions (or even within the same process definition), it can help you to generalize it so that it is indeed reusable. For example, rather than defining a service called “Credit Client Account”, you might decide to define it as “Adjust Client Account” where it can be called from different stages within a process to either credit or debit an account based on what value you pass into it.

Note: Refer to the “Process Development CeTN” - section “Reusable Services and Subprocesses” for more detailed explanations and examples.

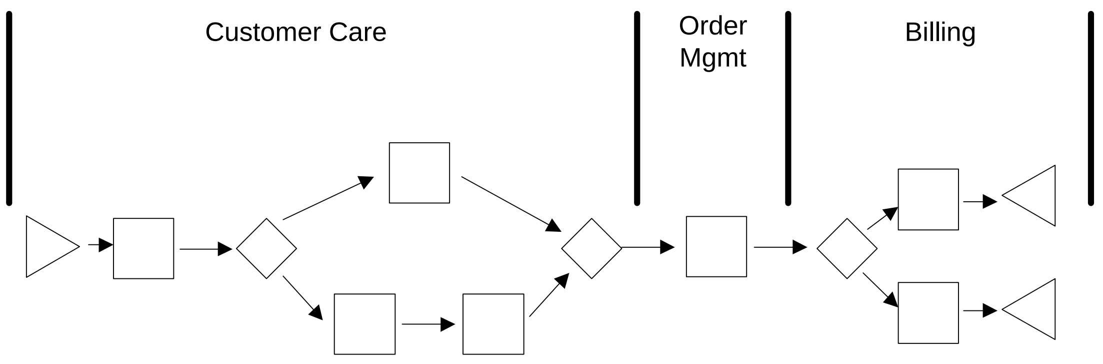
It is these Changengine service definitions that determine what functions will need to be implemented on your Middleware Bus.

For example, you might have a work node called “Bill The Customer” which calls the service “Billing Activation” (shown above), passing it `Cust ID` and `Order ID`, and expecting to get back the `Status`. You might decide that this is something to be implemented by your actual Billing system - hence you would need to do the necessary EAI work to provide this “Billing Activation” service on the Middleware Bus.

Remember, when defining your business process you want to try to imagine that you have a fully integrated set of applications and fully normalized data. You probably don’t have ...but... you want to assume that you do and then put together what the business process really should be - specifying the low level Changengine services that are required to do the job. These services then become your set of functions that must then be implemented on the Middleware Bus. These services effectively define exactly what integration you need across your application systems.

If we consider the earlier example Changengine process, the functional decomposition stages might look like this:

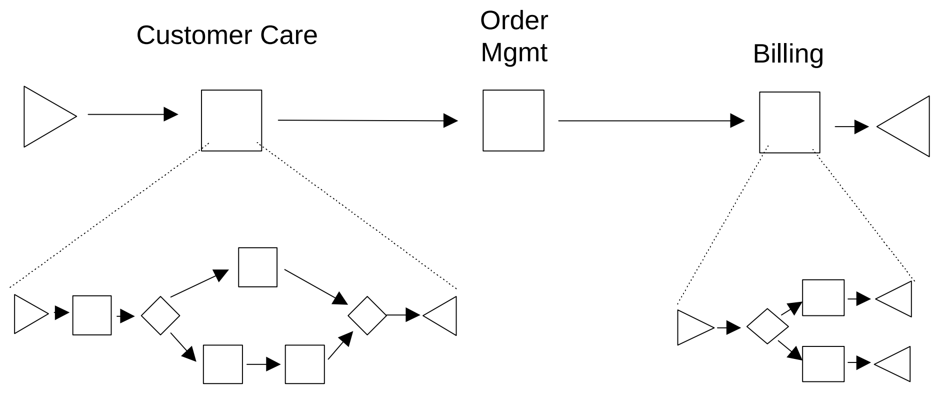
Identify some overall phases to the process:



It might be that these are phases and you just wish to annotate them within the Process Diagram - that’s fine. But it might be that you identify that some or all of these are actually common steps that will be repeated in future business processes. In which case you want to set these up as subprocesses, called from a top-level process.

(In this example, I am showing a fairly simple process diagram and it just happens to have three phases that match our three external application systems. Clearly this will not always be the case. I’m just trying to show the basic principles because they are essential when building good reusable business processes.)

Define the subprocesses:

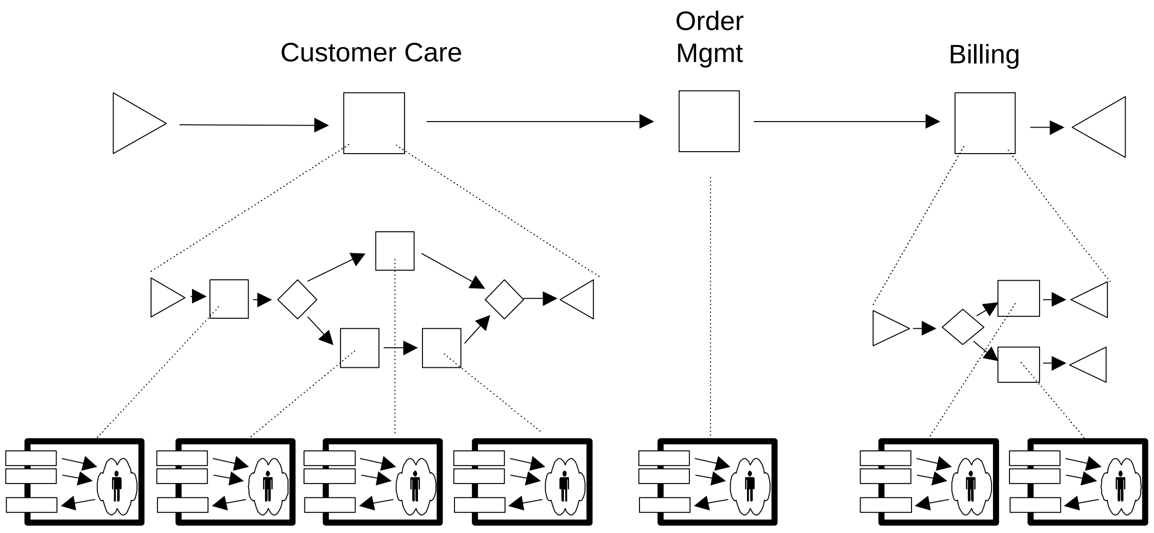


I’ve shown here that the overall process is in three main phases where both the Customer Care section and the Billing section break down into subprocesses. The Order Mgmt section is going to be reusable, but it only needs to be a single service - so there is no need to break it out into a subprocess.

What I’ve really done is say that the overall process breaks down into three globally reusable services - two of which are to be implemented as Changengine subprocesses.

This can often be difficult to understand at first, so I strongly recommend that you refer to the “Process Development CeTN” - section “Reusable Services and Subprocesses” for more detailed explanations and examples.

Define the services for each of these work nodes:



By the way, in this diagram I show every service looking exactly the same (2 inputs, 1 output) - this is just a pictorial representation :-)

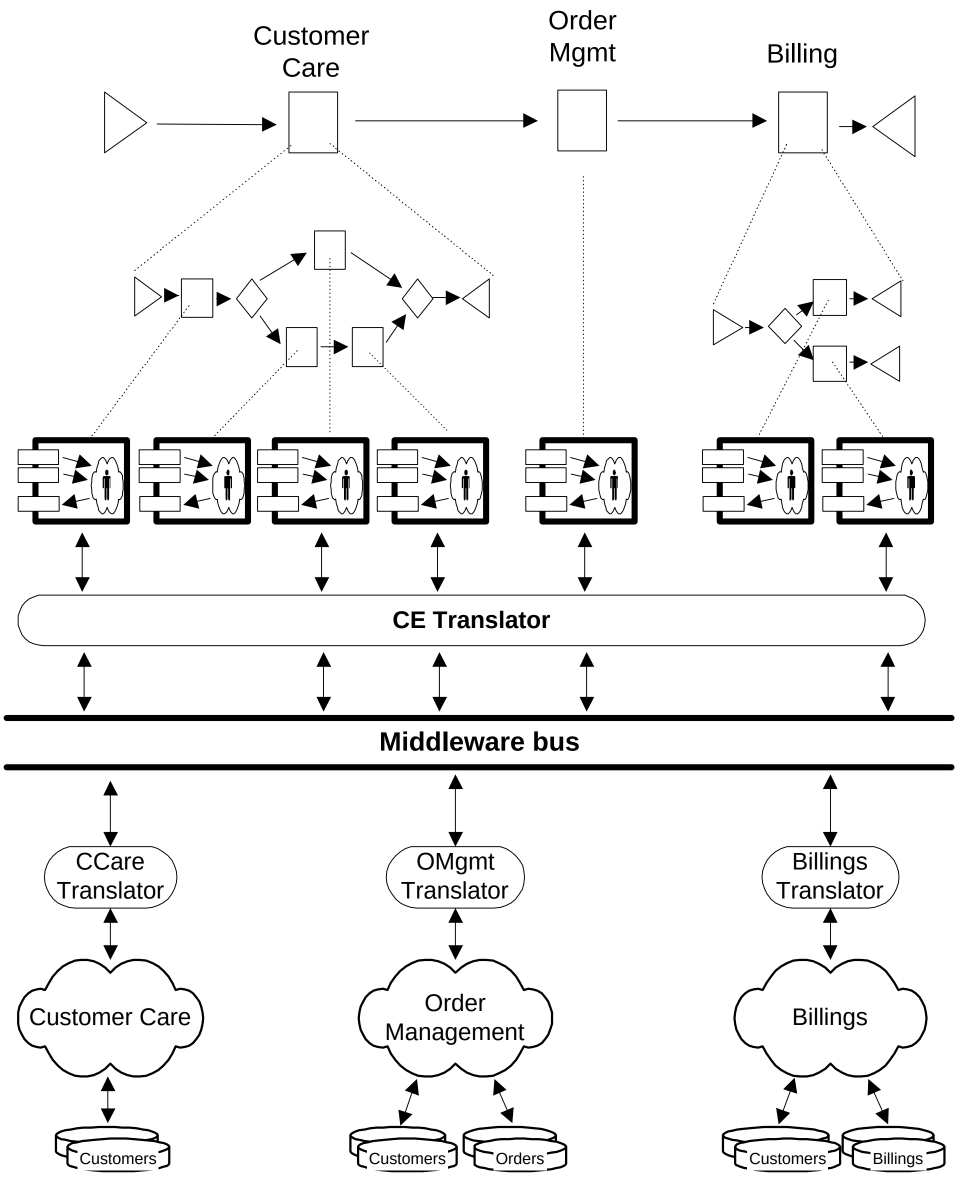
Obviously each service would be defined according to its needs - whatever inputs/outputs it required. But that is the whole point of this step - to specify the full list of services that we require to be implemented. To specify the inputs, outputs and the resource for each service, and to decide which of these are to be implemented across the Middleware Bus.

Map the services onto the Middleware Bus

Not every service is necessarily going to be performed by the underlying application systems. They may be, but I would expect to see a mix of services implemented by people and services implemented by applications. It all depends on the business process you are defining.

At this stage you have the list of services that are required to implement your business process. For each of these services you also have a clearly defined interface and specification of what the service must carry-out and what values (if any) it should return. For the services that are to be implemented via the underlying application systems you can now give these service definitions to the EAI team and this becomes their spec. for what functions they must make available across the Middleware bus.

When the EAI team have implemented these services you can run the process...and you can imagine that it would look something like this:



I've shown five of the services mapping to the Middleware bus. The other two are presumed to be implemented by sending the work item to a user. As I said earlier, it may be that all seven of these services map to the bus.

If you wanted to enable this whole process - the top-level process - to be startable from the application systems, then you would also need to map the start node's service to the Middleware bus. (I just don't show it on the diagram because I couldn't fit it all onto the page very easily :-)

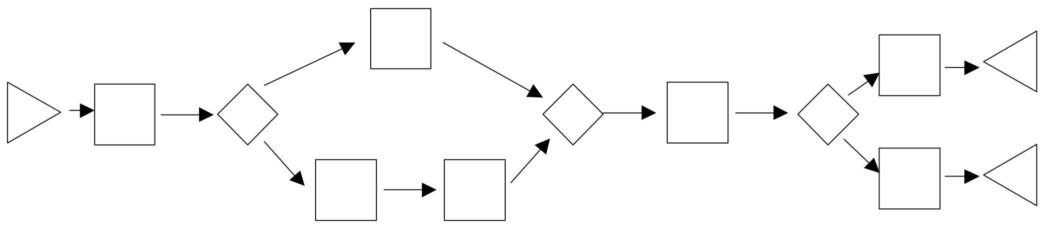
You have your business process defined. All reusable services and sections are in place for building future processes. You are bringing together the data and the intelligence of your underlying application systems. You now have a solution!

Additional Benefits

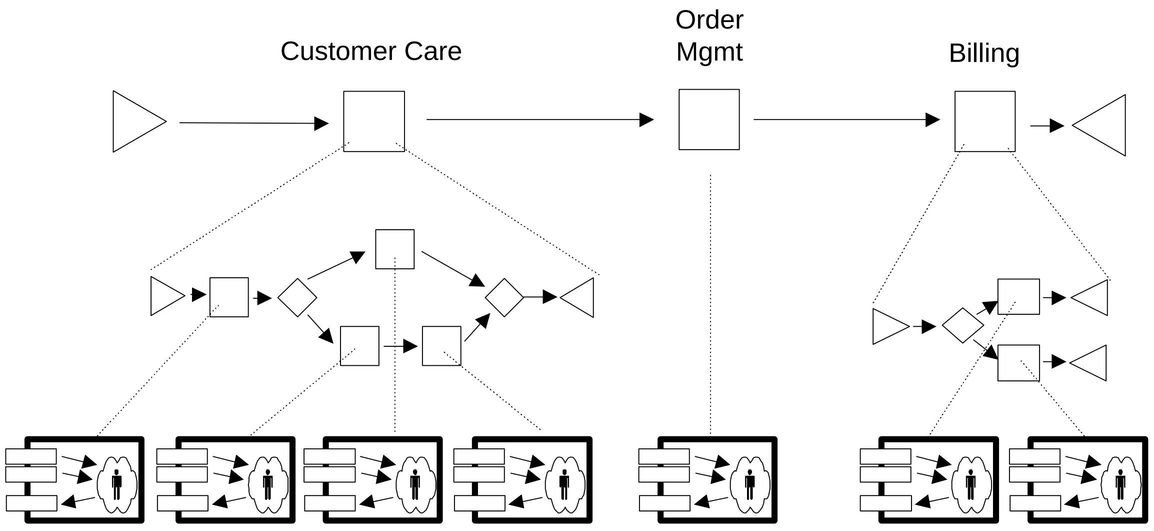
Whilst these next points are not necessarily to do with EAI, I mention them here because they are benefits that arise from functionally decomposing your business process in the way just described.

Process Metrics

If you think of our end-to-end business process, it started out looking like this:



and after breaking it down into reusable and functional areas, it now looks like this:



This has actually given us the ability to use the Changengine Business Console to give us metrics on our top-level process. This can show us straight away where we spend most of our time - in Customer Care, Order Management or Billing!

If we then wish to drill down to one of these subprocesses we can analyze that.

So, by subdividing the overall process into subprocesses we instantly gain better reporting.

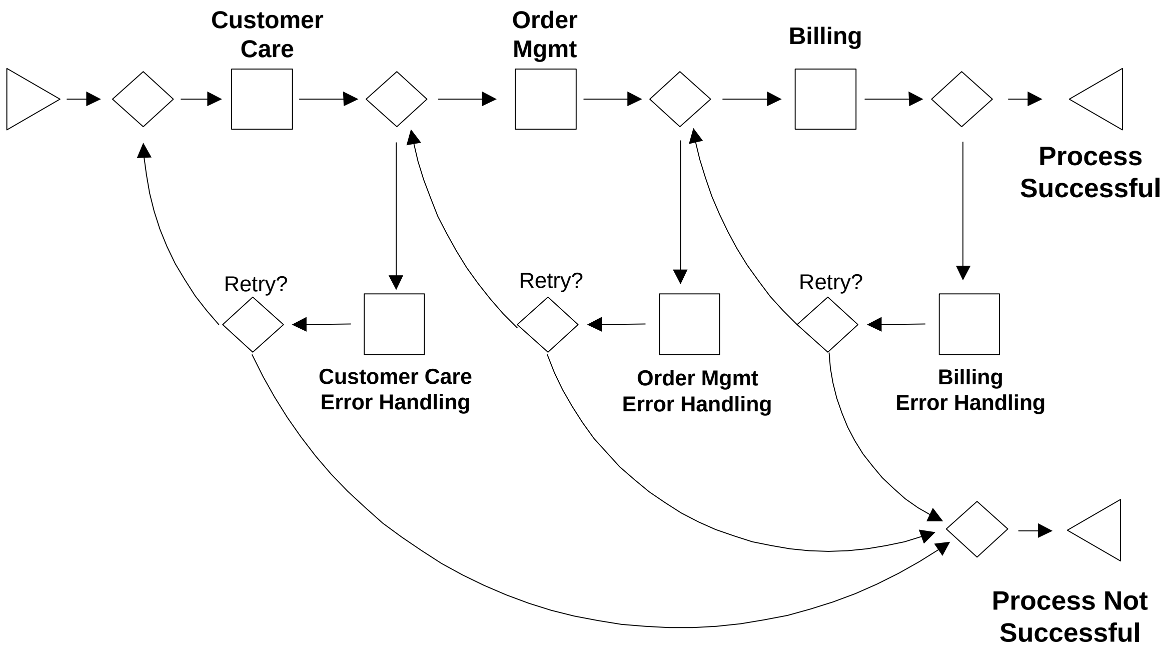
Don’t forget, the Changengine Audit Logs are just a set of SQL data tables/views, so if the standard Business Console metrics reports are not specific enough for you, you can always expand it with your own. You could also make use of packages such as Crystal Reports.

Error Handling

By breaking down the overall process into its subparts we also make it easier to put in error handling.

Obviously you can put a test after every single work node in every single process and test the state for success or failure, but your process diagram typically becomes unreadable and it is usually not necessary to go to that level.

What you probably want as a minimum is basic testing after each major phase. So your top level process could become:



Where we basically put a error detection loop after each of the main phases of the process (Customer Care, Order Management, Billing). The “Error Handling” node after each phase would initiate whatever cleanup actions were necessary and then either exit or retry.

Obviously it is up to you to define what these error handling cleanups must be, but it does mean that you can define these steps and have the process automatically catch errors and apply the clean up. You would also ensure that the service called within the Customer Care, Order Management and Billing nodes returned both a status and error text so that you could offer good reporting of the error.

By breaking the overall process down into reusable and functional areas it does lend itself to simple, effective and logical error handling.

Summary

You can see that to create a business process solution that spans integrated application systems is an involved, but very rewarding, effort.

There are clearly two major parts to this:

1. Defining the business process

   The business process will determine the scope of the integration effort that is to be involved, and determine the specification for the services (functions) that need to be made available on the Middleware bus.

   This drives the integration effort.

2. Integrating the application systems

   You need to choose the Middleware technology that you will use, the translators (adapters) that you will need and whether you will need to write these or buy them.

   You will need to come up with a normalized data model and then proceed to integrate the application system so as to provide the functions required to satisfy the business process.

The process definition team and the application integration team could carry out a lot of their work in parallel, but there would certainly be a lot of healthy discussion between the two. In particular, if the application team felt that they could not provide a certain service (function) for the process team then there would need to be discussion, and both teams would need to see what alternatives could be worked out.

Remember:

It is the business process that drives the application integration!