SIX AXES OF DECOUPLING

By: Dave McComb

semantic arts

Loose coupling has been a Holy Grail for systems developers for generations.

The virtues of loose coupling have been widely lauded, yet there has been little description about what is needed to achieve loose coupling. In this paper we describe our observations from projects we've been involved with.

Coupling

Two systems or two parts of a single system are considered coupled if a change to one of the systems unnecessarily affects the other system. So for instance, if we upgrade the version of our database and it requires that we upgrade the operating system for every client attached to that database, then we would say those two systems or those two parts of the system are tightly coupled.

Coupling is widely understood to be undesirable because of the spread of the side effects. As systems get larger and more complex, anything that causes a change in one part to affect a larger and larger footprint in the entire system is going to be expensive and destabilizing.

Loose Coupling/Decoupling

So, the converse of this is to design systems that are either "loosely coupled" or "decoupled." Loosely coupled systems do not arise by accident. They are intentionally designed such that change can be introduced around predefined flex points.

For instance, one common strategy is to define an application programming interface (API) which external users of a module or class can use. This simple technique allows the interior of the class or module or method to change without necessarily exporting a change in behavior to the users.

The Role of the Intermediate

In virtually every system that we've investigated that has achieved any degree of decoupling, we've found an "intermediate form." It is this intermediate form that allows the two systems or subsystems not to be directly connected to each other.

As shown in Figure (1), they are connected through an intermediary. In the example described above with an API, the signature of the interface is the intermediate.

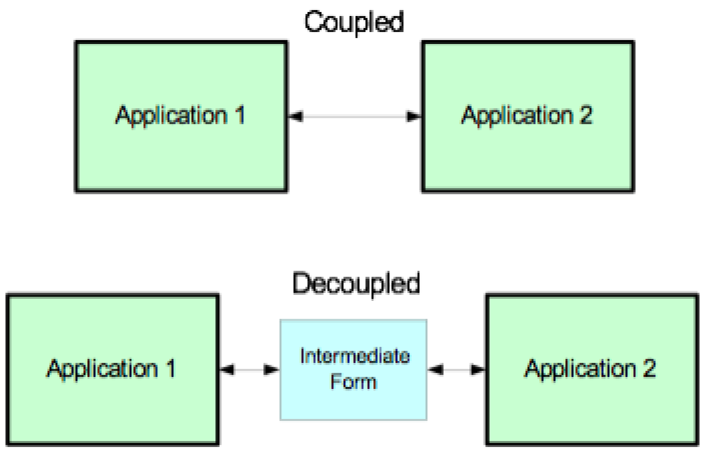


Figure 1

What Makes a Good Intermediary?

An intermediary needs several characteristics to be useful:

It doesn't change as rapidly as its clients. Introducing an intermediate that changes more frequently than either the producer or consumer of the service will not reduce change traffic in the system. Imagine a system built on an API which changes on a weekly basis. Every producer and consumer of the services that use the API would have to change along with the API and chaos would ensue.

It is nonproprietary. A proprietary intermediary is one that is effectively owned and controlled by a single group or small number of vendors. The reason proprietary intermediaries are undesirable is because the rate of change of the intermediary itself has been placed outside the control of the consumer. In many cases to use the service you must adopt the intermediary of the provider. It should also be noted that in many cases the controller of the proprietary standard has incentive to continue to change the standard if that can result in additional revenue for upgrades and the like.

It is evolvable. It's highly unlikely that anyone will design an intermediate form that is correct for all time from the initial design. Because of this, it's highly desirable to have intermediate forms that are evolvable. The best trait of an evolvable intermediate is that it can be added on to, without invalidating previous uses of it. We sometimes more accurately call this an accretive capability, meaning that things can be added on incrementally. The great advantage of an evolvable or accretive intermediary is that if there are many clients and many suppliers using the intermediary they do not have to all be changed in lockstep, which allows many more options for upgrade and change.

It is simple to use. An intermediate form that is complex or overly difficult to use will not be used and either other forms will be adopted which may be more various and different or the intermediate form will be skipped altogether and the benefit lost.

Shared Intermediates

In addition to the simple reduction in change traffic from having the intermediate be more stable than the components at either end, we also have an advantage in most cases where the intermediate allows re use of connections. This has been popularized in the Systems Integration business where people have pointed out time and time again that creating a hub will drastically reduce the number of interfaces needed to supply a system.

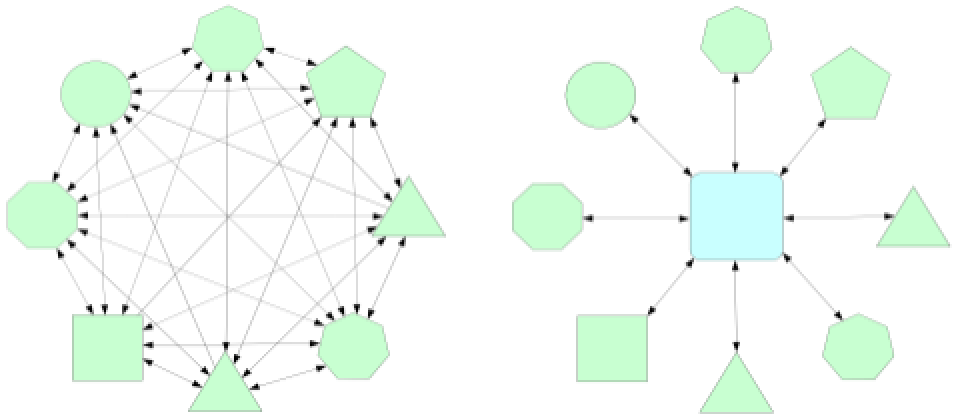


Figure 2

In Figure (2), we have an example of what we call the traditional interface math, where the introduction of a hub or intermediate form can drastically reduce the number of interconnections in a system.

People selling hubs very often refer to this as: (n * n – 1) / 2 or sometimes simply the n2 problem. While this makes for very compelling economics, our observation is that the true math for this style of system is much less generous but still positive. Just because two systems might be interconnected does not mean that they will be. Systems are not completely arbitrarily divided and therefore not every interconnection need be accounted for.

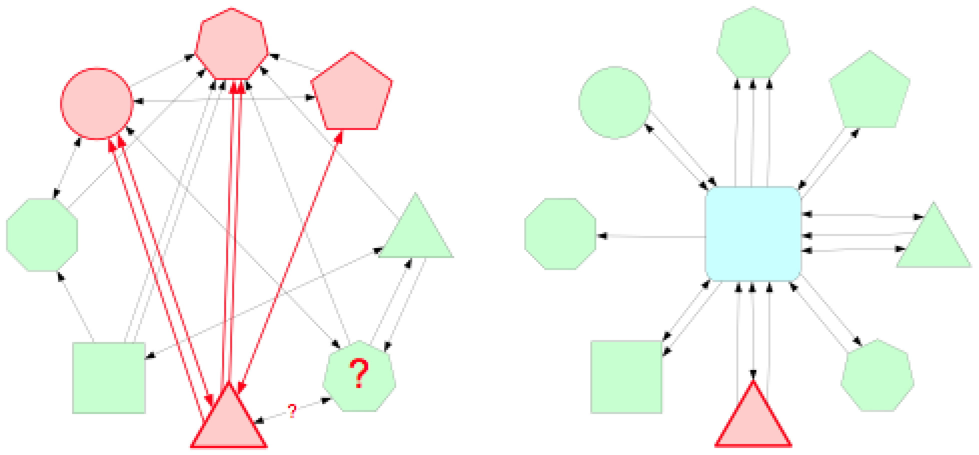


Figure 3

Figure (3) shows a more traditional scenario where, in the case on the left without a hub, there are many but not an exponential number of interfaces between systems. As the coloring shows, if you change one of those systems, any of the systems it touches may be affected and should at least be reviewed with an impact analysis. In the figure on the right, when the one system is changed, the evaluation is whether the effect spreads beyond the intermediary hub in the center. If it does not, if the system continues to obey the dictates of the intermediary form, than the change effect is, in fact, drastically reduced.

The Axes of Decoupling

We found in our work that, in many cases, people desire to decouple their systems and even go through the effort of creating intermediate forms or hubs and then build their systems to connect to those intermediate forms. However, as the systems evolve, very often they realize that a change in one of the systems does, in fact, "leak through" the abstraction in the intermediate and affects other systems.

In examining cases such as this, we have determined that there are six major considerations that cause systems that otherwise appear to be decoupled to have a secret or hidden coupling. We call these the axes of decoupling. If a system is successfully decoupled on each of these axes, then the impact of a change in any one of the systems should be greatly minimized.

Technology Dependency

The first axis that needs to be decoupled, and in some ways the hardest, is what we call technology dependency. In the current state of the practice, people attempt to achieve integration, as well as economy of system operation, by standardizing on a small number of underlying technologies, such as operating systems and databases. The hidden trap in this is that it is very easy to rely on the fact that two systems or subsystems are operating on the same platform. As a result, developers find it easy to join a table from another database to one in their own database if they find that to be a convenient solution. They find it easy to make use of a system function on a remote system if they know that the remote system supports the same programming languages, the same API, etc.

However, this is one of the most pernicious traps because as a complex system is constructed with more and more of these subtle technology dependencies, it becomes very hard to separate out any portion and re-implement it.

semantic arts

The solution to this, as shown in Figure (4), is to introduce an intermediate form that ensures that a system does not talk directly to another platform. The end result is that each application or subsystem or service can run on its own hardware, in its own operating system, using its own database management system, and not be affected by changes in other systems. Of course, each system or subsystem does have a technological dependency on the technology of the intermediary in the middle. This is the trade-off; you introduce the dependence on one platform in exchange for being independent of n other platforms. In the current state-of-the-art, most people use what's called an integration broker to achieve this. An integration broker is a product such as IBM's WebSphere or TIBCO or BEA, which allows one application to communicate with another without being aware of, or care, what platform the second application runs on.

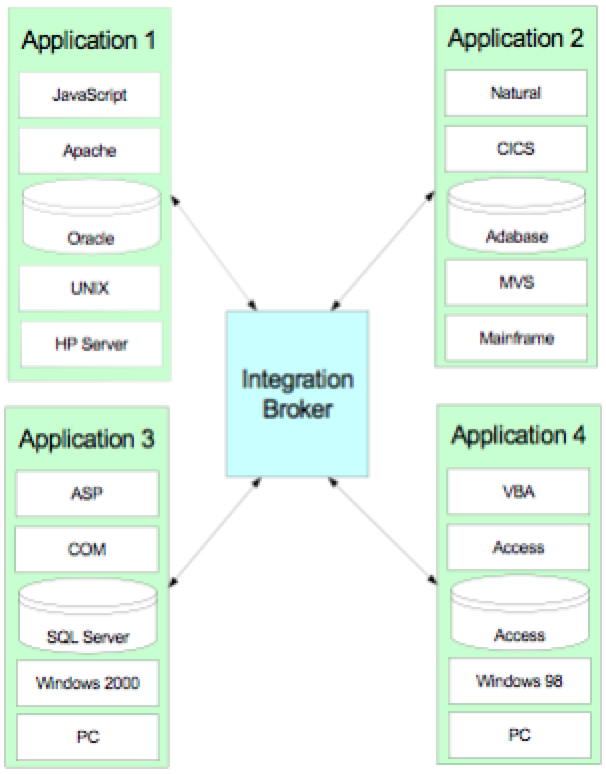
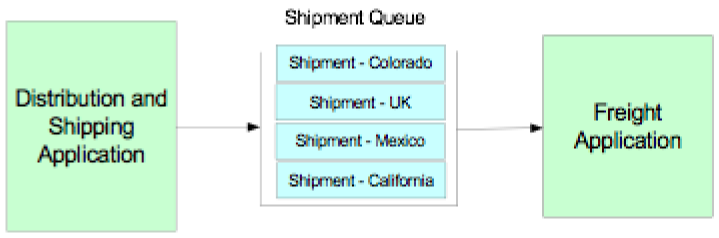


Figure 4

Destination Dependency

Even when you've successfully decoupled the platforms the two applications rely on, we've sometimes observed problems where one application "knows" of the existence and location of another application or service. By the way, this will become a very "normal problem" as Web services become more popular because the default method of implementing Web services has the requester knowing of the nature and destination of the service.



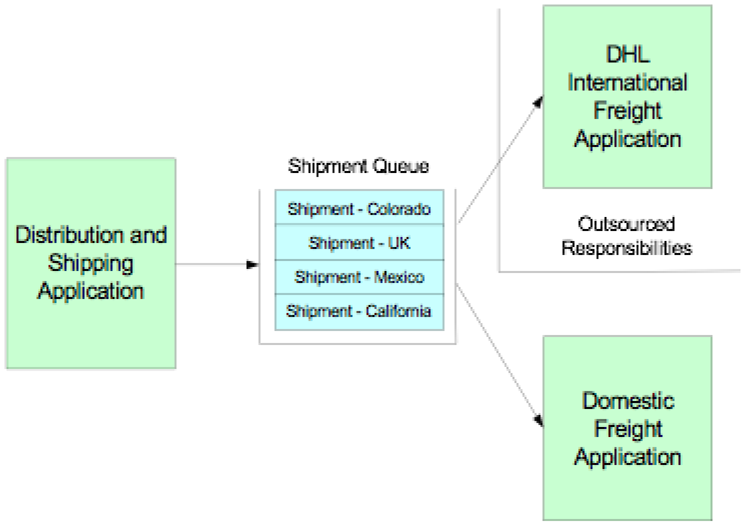
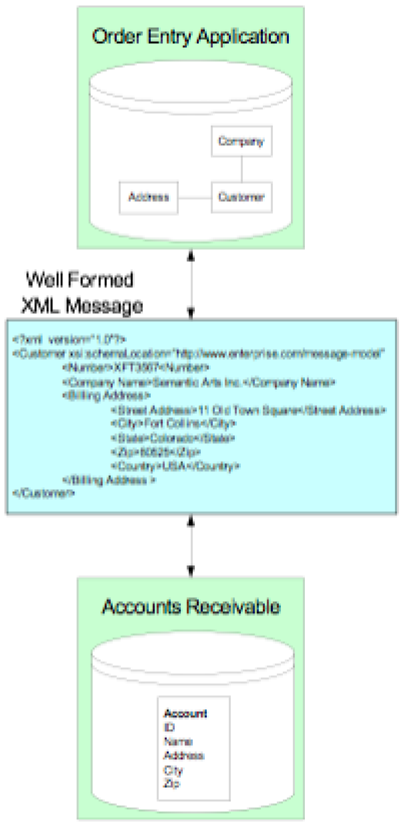

Distribution and Shipping Application
Shipment Queue
Shipment - Colorado
Shipment - UK
Shipment - Mexico
Shipment - California
DHL International Freight Application
Outsourced Responsibilities
Domestic Freight Application

Figure 5

In Figure (5), we show a little more clearly through an example where two systems have an intermediary. In this case, the distribution and shipping application would like to send messages to a freight application, for instance to get a freight rating or to determine how long it would take to get a package somewhere. Imagine if you were to introduce a new service in the freight area that in some cases handled international shipping, but we continue to do domestic the old way. If we had not decoupled these services, it is highly likely that the calling program would now need to be aware of the difference and make a determination in terms of what message to send, what API to call, where to send its request, etc. The only other defense would be to have yet another service that accepted all requests and then dispatched them; but this is really an unnecessary artifact that would have to be added into a system where the destination intermediary had not been designed in.

Syntax Intermediary

Classically in an API, the application programming interface defines very specifically the syntax of any message sent between two systems. For instance, the API specifies the number of arguments, their order, and their type; and any change to any of those will affect any of the calling programs. Also EDI (electronic data interchange) relies very much on a strict syntactical definition of the message being passed between partners.



```
<?xml version="1.0"?>
<Customer xsi:schemaLocation="http://www.enterprise.com/message-model"
        <Number>XFT3567</Number>
        <Company Name>Semantic Arts Inc.</Company Name>
        <Billing Address>
                <Street Address>11 Old Town Square</Street Address>
                <City>Fort Collins</City>
                <State>Colorado</State>
                <Zip>80525</Zip>
                <Country>USA</Country>
        </Billing Address >
</Customer>
```

Figure 6

In Figure (6), we show a small snippet of XML, which has recently become the de facto syntactic intermediate form. Virtually all new initiatives now use XML as the syntactic lingua franca. As such, any two systems that communicate through XML at least do not have to mediate differences at that syntactic level. Also, fortunately, XML is a nonproprietary standard and, at least to date, has been evolving very slowly.

Semantic Intermediary

Where systems integration projects generally run into the greatest amount of trouble is from semantic differences or ambiguities in the meaning of the information being passed back and forth. Traditionally, we find that developers build interfaces and run them and test them against live data, and then find that the ways in which the systems have been used does not conform particularly well to the spec. Additionally, in each case the names and therefore the implied semantics of all the elements used in the interface are typically different from system to system and must be reconciled. The n2 way of resolving this is to reconcile every system to every other system, a very tedious process.

There have been a few products and some approaches, as we show very simply and schematically in Figure (7), that have attempted to provide a semantic intermediary. Two that we're most familiar with are Condivo and Unicorn. Over the long-term, the intent of the Semantic Web is to build shared ontologies in OWL, which is the Web Ontology Language and a derivative of RDF and DAML+OIL. In the long-term, it's expected that systems will be able to communicate shared meaning through mutually committed ontologies.

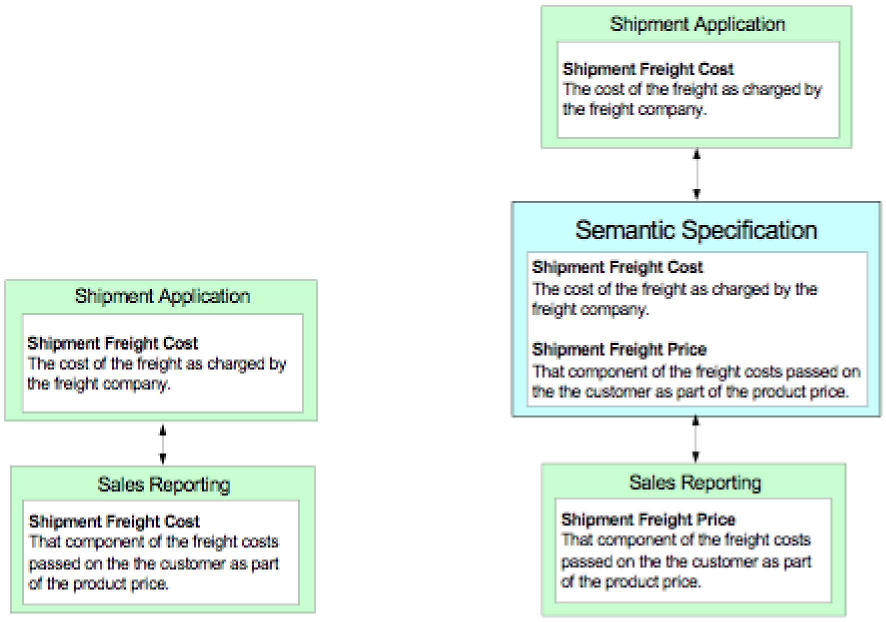

Shipment Application
Shipment Freight Cost
The cost of the freight as charged by the freight company.
Sales Reporting
Shipment Freight Cost
That component of the freight costs passed on the the customer as part of the product price.
Shipment Application
Shipment Freight Cost
The cost of the freight as charged by the freight company.
Semantic Specification
Shipment Freight Cost
The cost of the freight as charged by the freight company.
Shipment Freight Price
That component of the freight costs passed on the the customer as part of the product price.
Sales Reporting
Shipment Freight Price
That component of the freight costs passed on the the customer as part of the product price.

Figure 7

Identity Intermediary

A much subtler coupling that we've found in several systems is in the use of identifiers. Most systems have identifiers for all the key real world and invented entities that they deal with. For instance, most systems have identifiers for customers, patients, employees, sales orders, purchase orders, production lines, etc. All of these things must be given unique unambiguous names. That is not the problem; the problem is that each system has a tendency to create its own identifiers for items that are very often shared. In the real world, there is only one instance of many of these items. There is only one of each of us as individuals, one each for each building, one each for each corporation, etc. And yet each system tends to create its own numbering system and when it discovers a new customer it will give it the next available customer number. In order to communicate unambiguously with the system that's done this, to date the two main approaches have been either to force universal identifiers onto a large number of systems or to store other people's identifiers in your own system. Both of these approaches are flawed and do not scale well. In the case of the universal identifier, besides having all the problems of attempting to get coverage on the multiple domains, there is the converse problem of privacy. Once people, for instance, are given universal identifiers it's very hard to keep information about individuals anonymous. The other approach of storing others' identifiers in your systems does not scale well because as the number of systems you must communicate with grows, the number of other identifiers that you must store also grows. In addition, there is the problem of being notified when any changes to these identifiers occur.

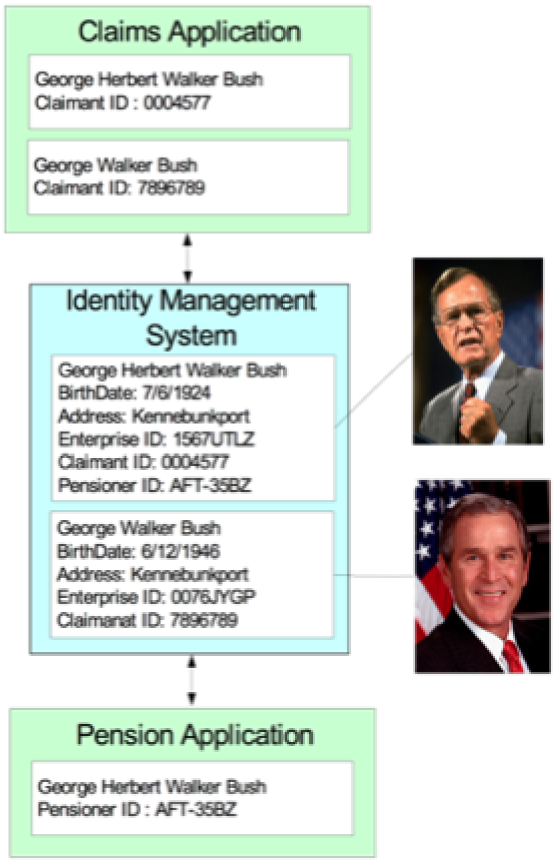


Figure 8

In Figure (8), we outline a new intermediary, which is just beginning to be discussed as a general-purpose service, variously called the identity intermediary or the handle intermediary. The reason we've begun shifting from calling it an identity intermediary is because the security industry has been referring to identity systems and it does not mean exactly the same thing as what we mean here. Essentially, this is a service where each subscribing system recognizes that it may be dealing with an entity that any of the other systems may have previously dealt with. So this has a discovery piece that systems can discover if they're dealing with, communicating with, or aware of any entity that has already been identified in the larger federation. It also acts as a cross reference so that each system need not keep track of all the synonyms of identifiers or handles to all the other systems. Figure (8) shows a very simple representation of this with two very similar individuals that need to be identified separately. To date, the only system that we know of that covers some of this territory is called ChoiceMaker, but it is not configured to be used in exactly the manner that we show here.

Nomenclature Intermediary

Very similar to the identity or handle intermediary is the nomenclature intermediary. We separate it because typically, with the identity intermediary, we're dealing with discovered real world entities and the reason we have synonyms is because multiple different systems are "discovering" the same physical real-world item.

In the case of the nomenclature intermediary system, we're dealing with an invented categorization system. Sometimes categorization systems are quite complex. In the medical industry we have SNOMED, HCPCS, and the CPT nomenclature. But also we have incredibly simple, and very often internally made up, classification systems, so in every case where we create a code file where we might have seven types of customer or orders or accidents or whatever that we tend to codify in order to get more uniformity, these are nomenclatures. What is helpful about having intermediary forms is that it enables multiple systems to either share or map to a common set of nomenclatures or codes.

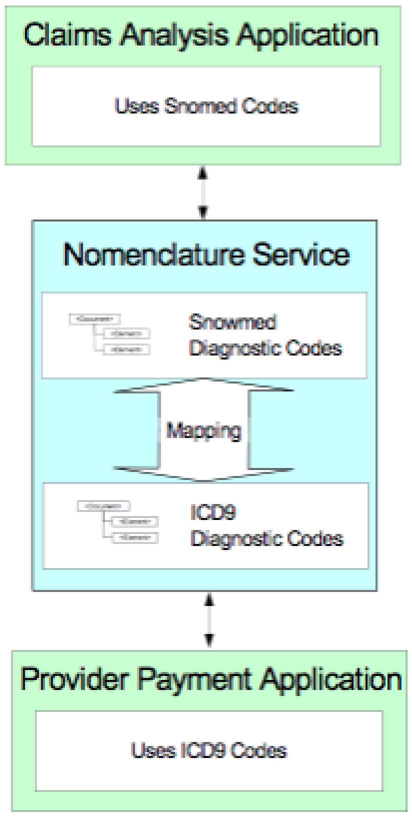


Figure 9

Figure (9) shows a simple case of how the mapping could be centralized. Again, this is another example where over the long-term, developments in Semantic Web may be a great help and may provide clearinghouses for the communication between disparate systems. In the meantime, the only example that we're aware of where a company has internally devoted a lot of attention to this is the Allstate Insurance Co., which has built what they call a domain management system where they have found, catalogued, and cross-referenced over 6,000 different nomenclatures that are in use within Allstate.

semantic arts

Summary

Loose coupling has been a Holy Grail for systems developers for generations. There is no silver bullet that will slay these problems; however, as we have discussed in this paper, there are a number of specific disciplined things that we can look at as developers, and as we continue to pay attention to these, we will make our systems more and more decoupled, and therefore easier and easier to evolve and change.

semantic arts

11 Old Town Square
Suite 200
Fort Collins, CO 80524

970-490-2224
305-425-2224
info@semanticarts.com