GETTING STARTED WITH DDD WHEN SURROUNDED BY LEGACY SYSTEMS

By
Eric Evans

GETTING DDD STARTED SURROUNDED BY LEGACY SYSTEMS

Attempts to employ Domain-Driven Design (DDD) tactics in the context of a legacy system almost always disappoint. In this paper, I'll describe three strategies for getting started with DDD when you have a big commitment to legacy systems. These strategies emphasize different goals and require different levels of organizational commitment.

STRATEGY 1: BUBBLE CONTEXT

1. **Why we need the bubble**
2. **How we form the bubble**
3. **What becomes of the bubble**

WHY WE NEED THE BUBBLE

We say that effectively applying the tactical techniques of DDD requires a clean, bounded context[1]. This can be a daunting requirement when your work is dominated by legacy systems. These systems are often tangled, and even when they are orderly they are usually not suited to DDD.

One of the fundamentals of DDD is that we choose a model (by which we mean a system of abstractions, not a UML diagram or other concrete artifact) well suited to the problem at hand. Yet a legacy system already has an established model, albeit implicit, and this model can seldom be changed with a reasonable amount of effort. Even if the legacy model could be changed, the new model might not suit the legacy functionality -- the change could undermine what the legacy system was always good at. On the other hand, simply adding objects that express a distinct model without changing the ones already there will lead to conflicting rules and concepts.

Then again, sometimes the old model is respected, the building blocks being intended as a way to bring order and expressiveness, rather than model transformation. Although in theory this could work, in practice it seldom succeeds because most such systems have multiple, intermingled models and/or the teams have disorderly development habits. These factors disrupt subtle design elements (especially in the case of a 'Big Ball of Mud'[2]).

This leads some organizations to introduce DDD with legacy replacement projects and other ambitious initiatives. I advise against this also. Such efforts are high-risk even for a team that already has mastered whatever techniques are to be employed. As I've

pointed out many times, legacy replacement is usually a bad strategy, only made worse by simultaneously introducing dramatic changes in process. Introducing a difficult new set of development principles and techniques is best done incrementally, as in a pilot project, in a way that allows members of the team to gain experience and allows the organization to assess the approach. The "bubble context" is a good candidate in this situation.

This first strategy does not require a big commitment to DDD. It allows even a small team to achieve a modest objective involving some intricate domain logic and, ideally, one with some strategic value. Then at some point the bubble bursts. The carefully designed code is gradually reabsorbed by the legacy. It is no longer a platform for innovative new development.

Yet the new functionality does not disappear. It will continue to be maintained as an extension of the legacy systems. The organization now has the experience to undertake more ambitious use of DDD.

Main Characteristics of the Bubble Context Strategy

- Modest commitment to DDD
- No synchronization risk (uses legacy database)
- Works when there is a limited range of data needed from legacy

HOW WE FORM THE BUBBLE

A 'bubble' is a small bounded context established using an Anticorruption Layer (ACL) (defined in my book[3] on p xxx and in the free downloadable *DDD Reference*[4]) for the purpose of a particular development effort and not intended necessarily to be expanded or used for a long time.

To get started, you need to choose an important, yet modest-sized, business related problem with some intricacy. You'll need a small, self-disciplined team (cherry-picked, if necessary) that has control over its code. The bubble isolates that work so the team can evolve a model that addresses the chosen area, relatively unconstrained by the concepts of the legacy systems.

Anticorruption Layer and Translation

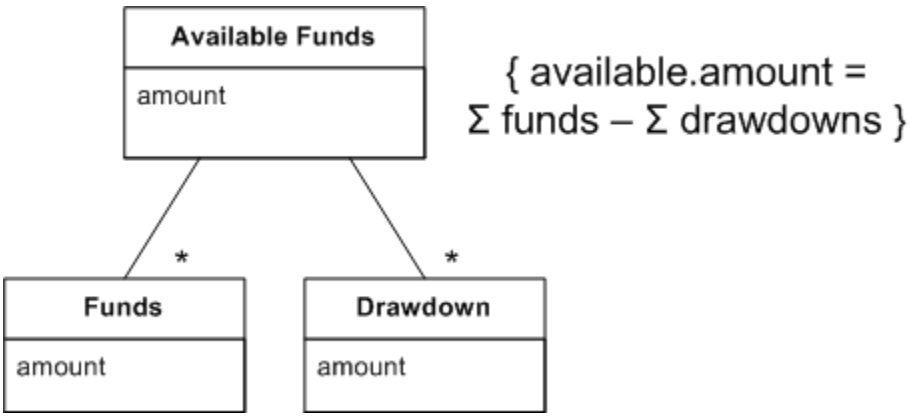
The context boundary of the bubble is established with the popular ACL.

Great Wall of China: A different sort of ACL

This boundary isolates your new work from the larger system, allowing you to have a very different model in your context than exists just on the other side of the border.

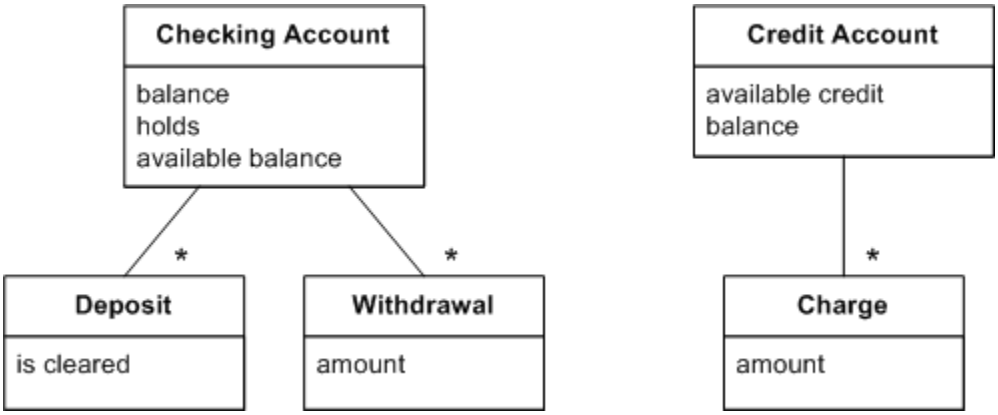
For an example, imagine our organization has two existing legacy systems, each of which tracks some customers' credit cards and checking accounts. Now we want to implement a feature called 'Can I buy it?', which figures out how much the customer could theoretically lay out for an impulse buy. This would be their balance in checking plus their credit card limit minus the charges (or 'drawdowns') on their cards. (Note that this feature is actually too simple to justify a bubble context in a real project. Examples have to be simple.)

Desired new model



Each of the legacy contexts defines some concepts which are mapped to the abstractions of our bubble context. First, a subsystem which we'll call "Context A" works according to this class diagram:

Existing model in Context A

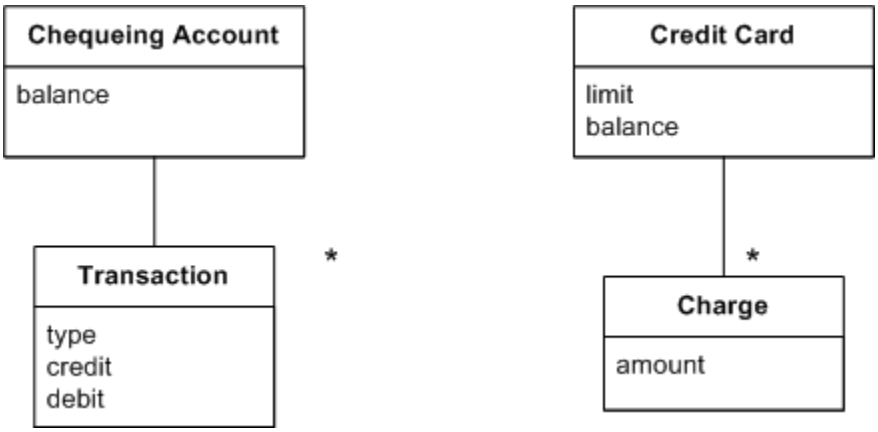


These objects can be mapped to our new model:

- CheckingAccount.available balance → Funds.amount
- CreditAccount.available credit → Funds.amount

Then "Context B" works according to this class diagram:

Existing model in Context B



These objects can also be mapped to our new model:

- CheckingAccount.balance → Funds.amount
- CreditCard.limit → Funds.amount
- CreditCard.balance → Drawdown.amount

Translation itself can be tricky. There are alternative ways to populate the new objects from the legacy data, and some of them might work as well as these. Others would be undesirable, even though they might produce the right numbers.

For example, in the following translation of Context A, the semantics don't match, even though the calculated AvailableFunds.amount would still be correct:

- CheckingAccount.balance → Funds.amount
- CheckingAccount.holds → Drawdown.amount

Both holds and drawdowns reduce the available, but that doesn't make them the same thing. This would make for a brittle design and poor communications with business stakeholders.

To illustrate another pitfall, the following translation of Context B would be semantically consistent, yet it would move business logic into the translator:

- CreditCard.limit - CreditCard.balance → Funds.amount

The relationship between funds, limit and balance should be modeled in a bounded context. There are gray areas, but ideally the translator should just translate.

So working out the translation is an important part of the analysis when using an anticorruption layer. The implementation of that translator can take many forms. As much as practical, it should be kept loosely coupled to the rest of the ACL.

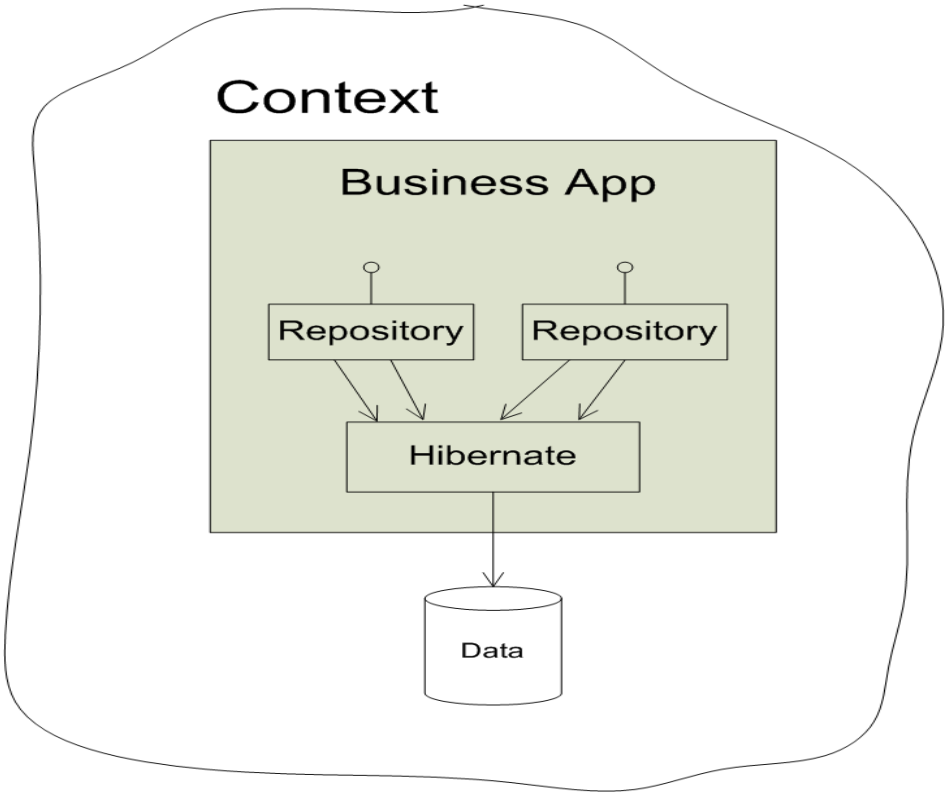
Now I'll turn to the interface of the ACL.

ACL-backed Repository
A nice technique that is sometimes used to set up a bubble context is the 'ACL-backed Repository'.

Remember, when we set up a bubble context, our data is coming from one or more legacy systems. We don't create a new database within the new context. We query the legacy database and rearrange the data into the new concepts in the ACL.

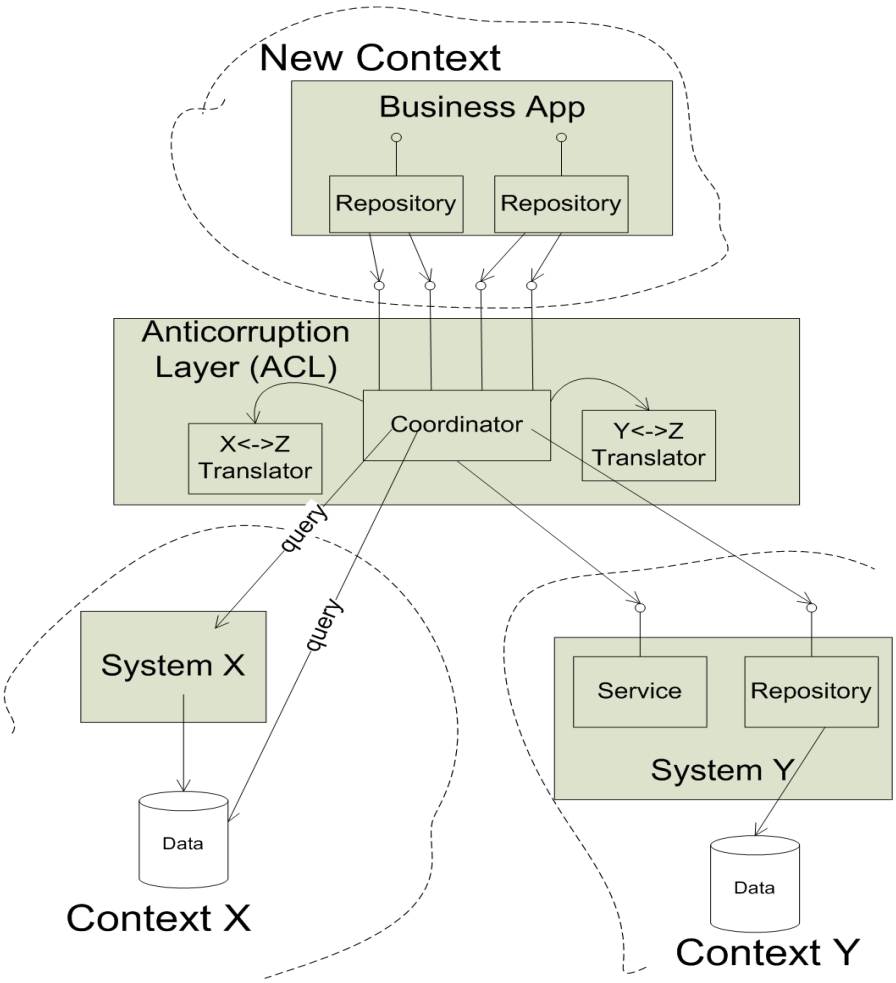
The DDD Building Block used to represent access to pre-existing objects is the 'Repository' (see Chapter 6, DDD[3] or DDD Reference[4].). We tend to think of a repository as a thin layer over a data access layer, which is mapping some kind of data store into our objects.

Classic Repository implementation



Yet, as with any design abstraction, the implementation of a repository can be entirely different from the impression given by the interface. We can create the illusion that the bubble context has its own data store by implementing a repository interface using the ACL. Any time an object is needed within the bubble, it is requested from the repository. The repository implementation in the ACL coordinates the steps needed to answer. First it invokes legacy functions and/or queries the legacy database(s). It passes the returned data to the translators, which rearrange the information into the objects used in the bubble. Finally, the ACL returns those objects through the repository interface. The repository's contract is fulfilled without having any data of its own.

ACL-Backed Repository implementation



It should be clear at this point that the anticorruption layer is a significant piece of software in its own right. It should be explicitly addressed in budgeting and planning. It should be designed explicitly as other components are, with particular care to keep it decoupled from the actual business logic of the new system. A good anticorruption layer has a clean interface entirely in terms of the downstream model (in our case the bubble model) that provides access to the information and services of the upstream system.

This means that development in the downstream context can focus on the business problem and not be distracted or complicated by dealing with the distinct concepts, and the quirks, of the other system. On the other hand, it also means that any refactoring of model elements where the information comes from the legacy context is extra work. We must refactor the new system and also the ACL. And any new information needed from the legacy context requires that we incorporate the new information into the bubble model, work out the mapping from the old model, and then enhance of the ACL. That's a high price, so we should choose functionality valuable enough to justify it.

WHAT BECOMES OF THE BUBBLE

Development in the bubble could go on for several months, or, occasionally, in cases where the interface with the other context is relatively narrow, for a few years. However, it will not last forever.

Sometimes, when the project has ended in success, people decide to make the new context the basis for more ambitious new development. When this happens, the limitations of the bubble context may be judged unacceptable, and the strategy migrates to the Autonomous Bubble with Synchronizing ACL (see below). Then the bubble transforms something more long-lasting.

More often, the bubble is temporary. Perhaps, as more is done in the new context, the discipline of the early work is not maintained. Holes appear in the ACL. The bubble bursts. Sometimes, with the initial work completed, interest dissipates. Neglect eventually leads to loss of the subtle design elements and bypassing of the ACL, and the bubble's isolation is gone.

Although most bubbles are short-lived, that does not mean they don't provide value. They allow a team and an organization to gain experience with DDD with a minimal commitment and low risk. They allow an intricate problem to be solved which would have been difficult to solve in the legacy context. And although the dissolution of the bubble means that tactical DDD will not be very helpful in changing or extending what was done, the functionality that was built in the bubble will live on long after the bubble is gone. As the boundary becomes less distinct, the fossilized model expression becomes just one of many in the mosaic of the legacy system.

The bubble context is a great way to get started with DDD while coexisting with legacy systems. In the coming months, I'll write about other strategies of varying levels of ambitiousness.

STRATEGY 2: AUTONOMOUS BUBBLE

1. Cutting the umbilical
2. The Synchronizing ACL
3. Bubble vs. Autonomous Bubble

CUTTING THE UMBILICAL

In the previous section I discussed the "Bubble Context" as a way to get started with DDD when there is not much organizational commitment. The bubble context is completely dependent on the parent context, drawing all its data through an anti-corruption layer (ACL) like an umbilical cord. You can use that strategy when little else might work. However, it has its limits.

Almost all software uses data that comes from other contexts. A lot of interesting domain logic revolves around combining data from different sources and reasoning about it. Even with one source, there may be a significantly new way of thinking about the data that is fundamental to the desired new functionality (a new way of abstracting the domain, i.e. a new domain model). Sometimes this is done in ways that tangle the systems together and prematurely limit design freedom. The bubble context is quite natural way to start with such an undertaking because the team can experiment with models and combinations in isolation from the legacy system, while the Umbilical ACL obtains and translates the data with minimal impact to existing systems.

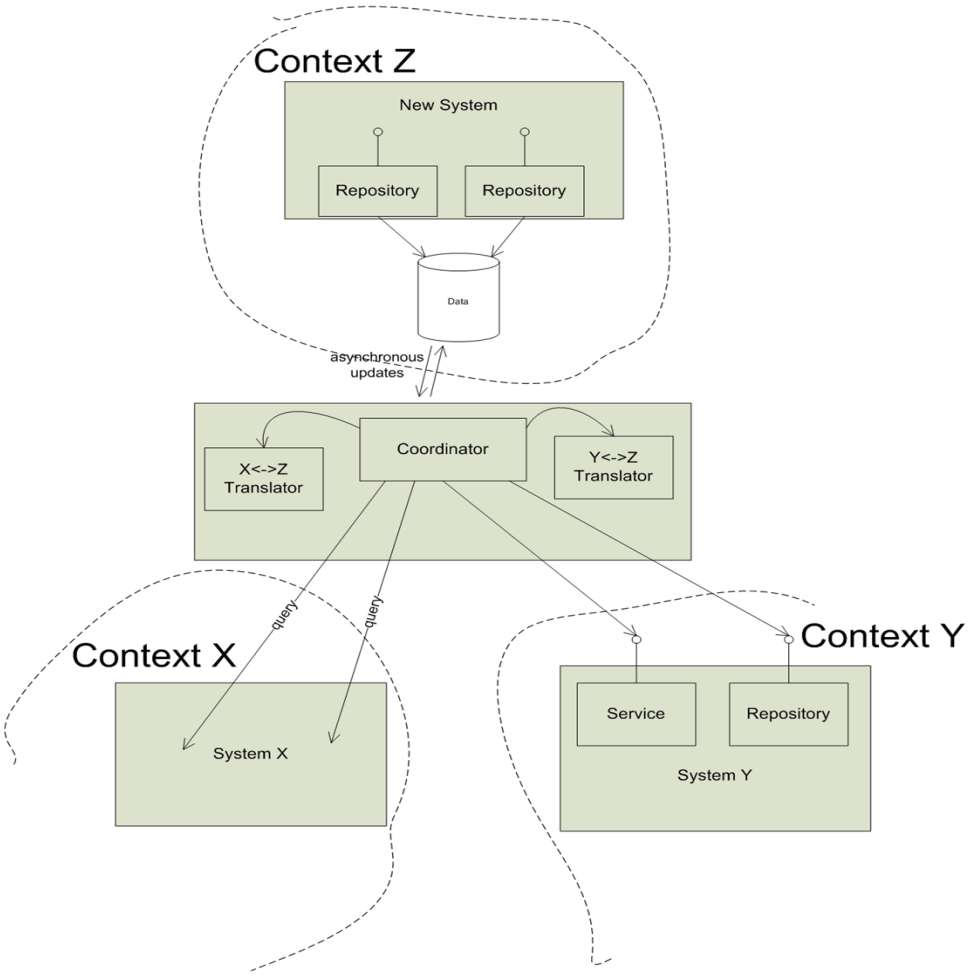
The Autonomous Bubble is distinguished by the ability to run its software, for a time, cut off from other systems. Information may start to get stale, requests to external service requests start to queue up unfulfilled, but the internal behavior of the software continues to work. Whereas the Bubble Context obtains its data from another context when it is needed, via the umbilical ACL, the Autonomous Bubble typically has some data store of its own. The data store could be in-memory — the storage medium is irrelevant — but the data is in the form native to the context. This is not just a cache of frequently or recently used data. In fact, an umbilical ACL could cache data as a performance enhancement. The point of the local data store is that the overall evolution of the model and design in this context is much more loosely coupled to other contexts. It should be possible to test such a subsystem without mocks of other subsystems.

Main Characteristics of the Autonomous Bubble Strategy

- Needs organization committed to significant new development with new design approach.
- Allows new data to be collected and used without involving legacy.
- Allows decoupled "product" to evolve.
- May progress from Bubble Context as development is scaled up on a successful pilot project.

SYNCHRONIZING ANTICORRUPTION LAYER
An autonomous context has its own data, sufficient to its core functions, organized according to in its own abstractions. To enable this, the ACL takes on the responsibility of synchronization between data stores in two contexts, which do not depend directly on each other. This ACL activity is asynchronous with any activity in either context, with a service level agreement (SLA) regarding the freshness of the translated data.

Synchronizing ACL



Compare this with the Umbilical ACL of the Bubble Context here

The Nightly Batch Script: Low-tech Synchronizing ACL
Asynchronous synchronization with an SLA sounds more daunting than it is. It can be quite low-tech and examples are all around us. A very familiar implementation of the Synchronizing ACL is the humble nightly batch.

Now a typical nightly batch script does all sorts of things, some of which include important business logic. But, in my observation, one thing most of them do is to update one data store based on another. This could involve importing data from a file. It could

involve kicking off a SQL script taking data from one table and transforming it and inserting it into another. It takes many forms, but look at a nightly batch script and you'll probably find a Synchronizing ACL mixed in there.

Typical nightly batch script has:

- SLA: Open of business, next business day
- Unnecessary coupling: Synchronizing ACL mixed in with once-per-day business logic
- Unnecessary coupling: Multiple ACLs to multiple contexts lumped together.

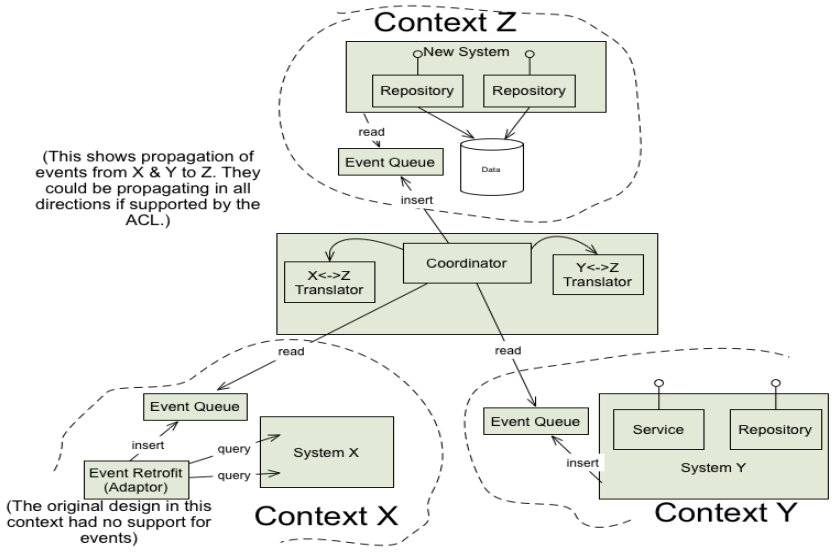
If you are trying to establish an autonomous bubble, and you think your translation could be handled as part of a nightly batch script, then make sure you isolate the data copying and translation for your particular context into modules (one per context you are drawing on) which can be kicked off by the script. Don't let it get tangled into the spagetti. You need a decoupled synchronized ACL to pull off an autonomous bubble.

Messaging and Domain Events: A Stylish Synchronizing ACL

A more flexible approach to synchronizing uses messages carrying Domain Events. This can be nicely integrated with other event-driven architectural patterns. This approach can support a range of SLAs, and can update a system while it is being used, while batch scripts typically run while systems are off-line.

People discuss the use of messages and events for integration quite often, but usually they don't talk explicitly enough about what happens when a message crosses a context boundary. Just remember, any information crossing a context boundary must be translated. Messages are not somehow neutral. They are always expressed in some language based on some system of abstractions (a model), and we should not let them enter a context that uses a different language and model.

ACL translating Domain Events



Many ways of implementing a Synchronizing ACL are in use and more will come in the future, so I won't try to catalog them. Any mechanism that can update a data store in one context based on data in another, translating along the way, and can do this asynchronously, and without introducing any dependency on it from either context, could be used to implement a Synchronizing ACL. You probably have Synchronizing ACLs in your system now, although they may be tangled with other responsibilities

BUBBLE VS. AUTONOMOUS BUBBLE

The first two strategies I've described each create a "bubble" context, and in this sense they are very similar, and resemble each other when looking inside the bubble. However, the difference in the relationship to the legacy contexts leads to a significant difference in the options for what you can do within the bubbles.

The "Umbilical ACL" from the first strategy gives us as little architectural overhead as possible, but allows no data to be used that can't be mapped from a context continuously attached. Any new information that does not already exist in the legacy system somewhere must be added to the legacy system somewhere, even though it will not be used there! Then it must be mapped into the bubble context, increasing the dependency between the two systems. Clearly this is not the strategy we want once we are introducing significant new information.

In contrast, the "Synchronizing ACL" gives more autonomy to the newly formed context. This autonomous bubble has its own data, some of which is mapped from another context, but it can also have data of its own. So development in the autonomous bubble is not so tightly bound to the older contexts. The model in this context can be more innovative and grow more freely, and a fresh new product can be created.

STRATEGY 3: EXPOSING LEGACY ASSETS AS SERVICES

1. **Why it is so hard to use legacy assets.**
2. **Open-host Service over ACL**
3. **Synchronizing or Backing ACL**
4. **Context Granularity**
5. **The Future of Legacy**

> *legacy*
> *Something inherited from a predecessor; a heritage: John Muir left as his legacy an enduring spirit of respect for the environment.*[5]

We strive for an architecture that lets us build on the work of the past, whether last month, last year or 10 years ago. For example, Service Oriented Architecture (SOA) promises the cultivation of suites of services that enable new applications to leverage preexisting functionality. Thus we could accumulate a wealth of preexisting functionality and pass that on to future development projects.

This turns out to be difficult, in practice. Even with relatively new software, designed with SOA in mind, there are design challenges that tend to be glossed over (most vitally, a lack of attention to bounded contexts). For a legacy system from a previous software generation, the idea of providing accessible services may seem inapplicable. However, it is often a good strategy.

One key to moving quickly is not to waste time recreating capabilities that already exist, especially in generic and supporting subdomains, discussed in chapter 15 of Domain-Driven Design [DDD]. Legacy systems often have great capabilities that the business depends on every day. Sometimes creating these capabilities originally involved a lot of grunt work, and repeating that grunt work is not a good use of time. These capabilities also involve integrations with yet more systems, which we'd rather not learn to talk to and certainly don't want to replace. Sometimes these capabilities, which have evolved over many years, reflect adaptations and nuances of the business that are not explicit or obvious, but which are nonetheless integral to business processes. These are some of the many reasons it is so difficult to replace legacy systems. We need to use them and concentrate our software development on the challenges of now.

WHY IT IS SO HARD TO USE LEGACY ASSETS

So we have good reasons for continuing to use these old systems, yet doing so holds us back in two specific ways. First, every new development effort that integrates with them gets dragged back into the old concepts and potentially tangled into the old implementation. This is the motivation for the bubble contexts of the previous sections. When we want to do something really *new*, legacy systems make us feel like we are wading through a vat of molasses, and bubble contexts can be a way to deliver high-value software quickly -- using anticorruption layers (ACLs) to access select data and functionality of the old system.

The second problem with legacy assets is how difficult and risky they are to invoke. They were developed with a particular feature-set in mind, and are highly coupled to those features. This makes it hard to access what you want directly without going through layers designed for very different purposes. The data you want is nearly always intertwined with other data and functionality, so that it is hard to detach just the part you want without dragging along an endless tangle of interactions and associations. Similarly, features we want to call typically have hard-to-predict side-effects - effects that were desired as a part of the business process for which the capability was originally developed, but which now take away its generality.

Each time we decide to call upon such a capability for some new purpose, someone has to meticulously sift through it all, finding an odd combination of inputs that will evoke the behavior or return the information we want without messing anything else up. It takes trial and error and extensive testing, and the result is typically brittle. The developer who made it work walks away exhausted, successful on his terms, leaving the system even more complex than before. The next time a closely related yet not identical need comes along, it starts all over.

Actually, this is very much the spirit in which we approach the development of the anticorruption layer of a bubble context. We are intent on enabling specific new features within the bubble, so we come up with a very specific adaptation to get the capability we need from the legacy, and it is typically messy and brittle. We carefully keep all this encapsulated within the ACL, away from the new development work. But we are not trying to lay the groundwork for other applications that might come after us. If someone else wants to use the legacy capability we tapped, they'll have to figure out their own way to trick the legacy system into giving them what they need.

OPEN-HOST SERVICE OVER ANTICORRUPTION LAYER

The Open-host Service [DDD[3], p 374 or Reference[4]] describes a general style of software interfaces in which we expose the capabilities of some software as a protocol. This pattern is the heart of SOA, yet also appears in many designs that have nothing to do with service frameworks. It is very common in modern systems. It is also a fairly common pattern for making the assets of a legacy system more accessible.

Example: Sales History

Our example company has many needs for historical sales information (by which I mean anything in the past, potentially including yesterday or earlier today). That data is all available in the database, and various features and reports have been built from it over the years. However, these features and reports are always complex to implement. For one thing, the sales are not all recorded in the same place. They are spread over three database tables used by two systems. Some of the entities in these tables are not even sales. Mixed in with the sales, two of the tables also contain orders that were cancelled before they were confirmed, and one of them contains shipments between warehouses for inventory balancing. Furthermore, the data tables were designed for supporting the processing of orders and are intertwined with that process. They contain, respectively, 12, 21 and 37 fields, and even so, at least one join is required to obtain some of the basic information about a sale (e.g. price per item).

The team defined a model for a stripped-down sale that would address the needs of several consuming applications, with basic "who, what, how much" information in a convenient, denormalized form. They defined a sale as a confirmed order from a customer, to eliminate the spurious entities such as inter-warehouse orders. Having clarified their concepts, they then defined a JSON schema that conformed to their model, and described a few basic queries (by date range, by customer, by item ID ...). So they had a design that was aligned with the model in the context of their service. Now they needed an implementation.

They wrote four moderately complex queries to select the data they needed, and a post-query filter to get rid of some non-sale records the queries still pulled in (a trade-off on the complexity of the query itself.) They wrote code that took data from these disparate query results and stitched it together into JSON records using the uniform representation. The query and code were a bit ugly and quirky - it dealt with many little quirks of the old schema they discovered through quick iterative cycles of testing and tweaking -- but eventually it worked. This query and code constituted the guts of an ACL. However, the output of their translator was different in style from that typical of a bubble ACL. It was a minimalist form digestible by several consuming subsystems, yet not meant to be used as an internal model for any of them.

ACL BACKED OR SYNCHRONIZING

Now that they had a translator, they faced an architectural choice very similar to the one covered in the previous two strategies: Whether to use an ACL backed Open Host Service (similar to ACL-backed Repository from strategy 1) or a Synchronizing ACL that populated a data store owned by the service (much like the Synchronizing ACL that populated the Autonomous Bubble's data store in strategy 2).

The tradeoffs are very similar. When the delivery of the service is in the form of a special technology like a web service or a REST /JSON interface, the synchronizing approach is favored pretty heavily because of the standardization of the technology, and it can give a big system performance boost in some cases. Where the service is to be used within the same execution platform, and is returning lists of ordinary objects, there are many cases where the on-demand calculation of the umbilical ACL-backed approach might be favored. In my experience, there tends to be less of a commitment to the decision between synchronizing and umbilical here than in the bubbles - it is easier to change your mind and make the service work the other way.

Note that the example service is a side-effect-free function [DDD p 250] and has all the usual attendant advantages. It is also common to need access to services with side-effects. For example, we might want a service to inject a sales record that originated elsewhere. Such a service would be attached to an anticorruption layer that would figure out how to insert it into the legacy database correctly. The approach described here is applicable to that case as well - everything is just a bit trickier because side-effects are tricky. The payoff of having a safe way to make such updates to the legacy can greatly ease future integrations and help prevent coupling of new development to the legacy design.

CONTEXT GRANULARITY

As with any well-defined model, the one underlying a service needs to have a context boundary. We might have defined a model for just one service, so that the single service constitutes the bounded context. There is nothing wrong with that, but if we have many services, it starts to add a lot of mental overhead of context shifting - definitions are shifting every time you use a different service. There is a lot of value in grouping services together into suites of services that share a lot of concepts or tend to be used together. If each of these suites has its own bounded context, this can be a practical level of granularity.

A pattern that tends to crop up about this point is one I call "The Commons Context". The notion is that the most fundamental and widely used information can be interchanged throughout the organization via one common interchange model. This is not the Enterprise Model - no application would use the commons model internally. Even so, this pattern is harder and more problematic than it appears. One reason I call it The Commons is that it can fall pretty easily to the "Tragedy of the Commons". The Commons is prone to dumping. Somehow, everything has to go in there, even if it hasn't been sharply modeled yet. And, whereas every consumer of the commons benefits from it being tidy, it is always most expedient to an individual consumer/extender to take shortcuts. For this reason, I generally favor contexts that give definition to smaller suites of services. Even so, a carefully managed, minimalist Commons, focused on very widely used data and entities, is enticing, and occasionally someone pulls it off.

A special case arises in some industries that have standardized on a "Published Language" [DDD p 375], and some of these are pretty comprehensive. An example of one of the most successful is Financial Information eXchange (FIX) Protocol[6]. FIX has a way of describing an enormous range of financial trades and transactions and is used to interchange data between all sorts of organizations. These comprehensive published languages are stable and well documented. Such a language might be a good option for a service that is to be accessed by software developed with no coordination or when the consuming software might already be able to consume the published language. However, these industry standards are designed by committee and are usually cumbersome. Their generality robs them of expressiveness. And, of course, you can't change them.

A focused language, crafted for a modest suite of related services, usually gives the most satisfying balance between the fragmentation of the one-off service model and the total unification of the Commons and the industry standard.

THE FUTURE OF LEGACY

By providing access to these capabilities in a separate context, we make it easier to change to a different implementation at a later time. At first, the owner of the data in question is the legacy system the ACL is dredging it from, and this may remain the case for a long time. Eventually, though, a new source may be developed (or multiple sources). The service may begin to draw its information from that new source, and the synchronization between the legacy system and the service might begin to flow the other way, with the legacy state being maintained by accessing the service (This being

necessary in order to support legacy behavior dependent on those resources). This general strategy does give us a piecemeal approach to phasing out legacy systems that incrementally improves the environment for new development at each point.

However, we should get over the attitude that the ultimate goal is to eventually replace these older systems. That is a futile treadmill. Creating a bubble can allow you to selectively move responsibilities for the core domain, where you want to innovate, out into a fresh context. Open-host services over an ACL can give convenient, decoupled access to functionality and data assets embedded in the older systems. Once a legacy system is stable, not responsible for core domain, and provides access to assets, why not let it be? You have better things to do right now.

In general usage, the word "legacy" carries many positive connotations. Examine what you have received for potential assets, and you will see many opportunities to focus your development on higher-value areas, given a reasonable investment in context boundaries and translators. Remember that if you do a good job, your software will be used for many years. You should be writing a legacy system right now.

STRATEGY 4: EXPANDING A BUBBLE

1. **Coevolving model and ACL**
2. **Only bring in data you use!**
3. **Adding data is a modeling job**

COEVOLVING MODEL AND ANTICORRUPTION LAYER (ACL)
In two of the previous articles in this series, I wrote about using 'bubble contexts' of various kinds to enable a fresh DDD project to coexist with a heavy legacy of older software.

Design freedom is the point of a bubble. The **bubble context with an umbilical ACL** gives a little freedom, while the **Autonomous Bubble** can give a lot, if properly managed. However, in the actual unfolding of a development project, there are many ways this can go wrong and bog down before the team has reached its goals. As soon as it becomes impossible to understand how a feature within the bubble works without reference to legacy concepts, then the context boundary is too weak for the typical goals of a bubble to be fulfilled. While modeling issues are fundamentally the same as in other contexts, working in a bubble context shifts the emphasis.

One challenge that is particularly acute in a bubble is how critical it is to coordinate the incremental development of the new business functionality and the anticorruption layer in parallel. This affects the design **and also the planning of the sequence in which the features are written**.

As features are chosen for development in a given iteration, in addition to the usual prioritization by business urgency or strategic value, the team needs to be thinking about what the implications will be for the anticorruption layer. Does the new feature draw on information that is already being brought in by the existing anticorruption layer? Great! Or will it require new mapping of legacy information? If so, then what is the scope of that work? If multiple features chosen for the iteration call for new mapping, are they drawing on the same information, or does each call for different information? To make development manageable, the priority of business features must be balanced with the need to group work on features that use similar legacy data sets, with parallel planning of the necessary enhancements to the anticorruption layer.

If you do not do this parallel planning, developers working on simple business features will get bogged down. The bulk of the project's effort will go into implementing a big translator interface, each element of which is only sparsely used. This gives a low ROI for the context boundary, and eventually an unsustainable maintenance burden. As soon as developers feel frustrated by this, the pressure they are under to finish their tasks will lead them to take shortcuts by making sloppy mappings or simply bypassing the ACL, pulling the data they need directly from the legacy system. Multiple developers will be doing this in parallel, and the context boundary will erode and collapse.

In contrast, when this work is planned more carefully, in any given iteration there will be some features where development either uses the anticorruption as-is or with a tweak to accommodate a downstream model refinement using the same old information. And there may also be some features that draw on a manageable new data set or feature that needs to be modeled in the new context and mapped in the software. The developers working on various features all drawing that same data can collaborate to:

1. Model the new information by adding concepts to the bubble context
2. Extend the ACL to bring in the new information. (This should be an explicit development task, not just part of a 'user story'.)

Only bring in data you use!
Obvious as this sounds, this rule is violated in most object-oriented systems I see, but it is a particular pitfall when working in a bubble.

As a typical example, suppose the business people, while viewing the results of the analysis provided by the new bubble functionality, also wish to see, side-by-side, some other information which is not currently imported into the context. The information is available as two fields in the legacy database, and people are familiar with the way it is expressed there, and fairly content with it. Of course, the easiest way to implement the view is to stick these two extra fields into an object being constructed in the anticorruption layer. The two fields would then be convenient to display along with other data and analysis represented by the objects in the new context.

This is a mistake. The objects will be bloated with information they don't need, and, worse still, the data in these fields will not have been translated! Sure, technically, they came through the translator, but the concept represented by those fields was never mapped to a concept in the downstream context. We have no idea **what they mean** in the downstream context or how they relate to other concepts in the downstream model. So we have a bigger object with more dependencies that is poorly defined. This is the stuff Big Balls of Mud are made of.

When you need that extra field (And there always is one. And then another one...) you must make a choice:

1. Extend the model.
   Make the bubble's model capable of expressing the essential meaning of the new information in terms consistent with the other concepts of that model. Even when you have translated, it is inadvisable to load objects down with information that does not directly pertain to their jobs, so it is best to do a bit more modeling work to find good homes for the new information.
2. Bypass the context altogether.
   Give the client access to the context where the information originated or pass it some other way. (This is my recommendation, in most cases, particularly when the new data is for display only.) Sometimes this is a good time to expose a legacy asset with a service as in Strategy 3.

3. Pass the information unchanged through your new context.
   (Not recommended) If you choose this lazy way, you have bored a tunnel through your anticorruption layer, and compromised your ability to evolve your new model independently of the legacy system.

Adding data is a modeling job

Once you have decided you really do want some new information in your context, you need to take this as a serious modeling and development task. It isn't "just a field". Over in the legacy system, that field means something (even if it is a bit muddled). It is a way of referring to some aspect of some business situation, and the goal when translating this new element into your bubble context is to enrich the ubiquitous language to be able to express those concerns. This calls for the same creative process we would use to explore new scenarios in the DDD Modeling Whirlpool[7].

In some ways, the challenge is greater than dealing with completely new concepts because it is so difficult to break the preconceptions carried over from the legacy context. Also, you have to fight the perception that this is "done". Once again, be skeptical about a decision to pull in more data. If you are using it in the logic or key definitions of your context, you should follow the whirlpool steps:

1. Hammer out a reference scenario that illustrates why you need the new information.
2. Brainstorm model changes that incorporate that information organically.

If you end up simply adding fields that map directly to legacy fields, then possibly it means those concepts of the old model are still the right fit for the business at hand. More likely it is a copout.

Some conceptual leakage is inevitable. Some blind spots created by familiarity with existing solutions are inescapable. Don't be paralyzed by perfectionism. But be wary when people are adding information into a bubble context without even seriously trying to synthesize a new model. Context boundary leakage gets out of hand quickly, and suddenly you may wake up to realize that your new software is conforming to the legacy model and does not give the desired fresh start for the business.

---

[1] "Four Prerequisites for DDD", Sept. 2010, < http://domainlanguage.com/newsletter/2010-09/>.

[2] "Big Ball of Mud", June 1999, Foote, Yoder < http://www.laputan.org/mud/>.

[3] *Domain-Driven Design, Tackling Complexity in the Heart of Software*, Evans, 2004.

[4] *DDD Reference*, 2011, < http://domainlanguage.com/ddd/patterns/>.

[5] Wiktionary, <http://en.wiktionary.org/wiki/legacy>.

[6] FIX, <http://fixprotocol.org/>

[7] "Model Exploration Whirlpool", <http://domainlanguage.com/ddd/whirlpool/>.