From: http://www.methodsandtools.com/archive/archive.php?id=97

# An Introduction to Domain Driven Design

*Dan Haywood*, Haywood Associates Ltd, http://danhaywood.com/

Today's enterprise applications are undoubtedly sophisticated and rely on some specialized technologies (persistence, AJAX, web services and so on) to do what they do. And as developers it's understandable that we tend to focus on these technical details. But the truth is that a system that doesn't solve the business needs is of no use to anyone, no matter how pretty it looks or how well architected its infrastructure.

The philosophy of **domain-driven design** (DDD) – first described by Eric Evans in his book [1] of the same name – is about placing our attention at the heart of the application, focusing on the complexity that is intrinsic to the business domain itself. We also distinguish the core domain (unique to the business) from the supporting sub-domains (typically generic in nature, such as money or time), and place appropriately more of our design efforts on the core.

Domain-driven design consists of a set of patterns for building enterprise applications from the domain model out. In your software career you may well have encountered many of these ideas already, especially if you are a seasoned developer in an OO language. But applying them together will allow you to build systems that genuinely meet the needs of the business.

In this article I'm going to run through some of the main patterns of DDD, pick up on some areas where newbies seem to struggle, and highlight some tools and resources (one in particular) to help you apply DDD in your work.

## Of Code and Models…

With DDD we're looking to create models of a problem domain. The persistence, user interfaces and messaging stuff can come later, it's the domain that needs to be understood, because that's the bit in the system being built that distinguishes your company's business from your competitors. (And if that isn't true, then consider buying a packaged product instead).

By model we don't mean a diagram or set of diagrams; sure, diagrams are useful but they aren't the model, just different *views* of the model (see Figure). No, the model is the set of concepts that we select to be implemented in software, represented in code and any other software artifact used to construct the delivered system. In other words, the code is the model. Text editors provide one way to work with this model, though modern tools provide plenty of other visualizations too (UML class diagrams, entity-relationship diagrams, Spring beandocs [2], Struts/JSF flows, and so on).

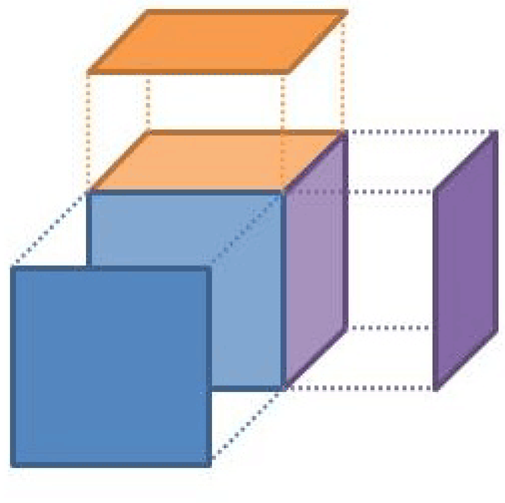


Figure 1: Model vs Views of the Model

This then is the first of the DDD patterns: a ***model-driven design***. It means being able to map – ideally quite literally – the concepts in the model to those of the design/code. A change in the model implies a change to the code; changing the code means the model has changed. DDD doesn’t mandate that you model the domain using object-orientation – we could build models using a rules engine, for example – but given that the dominant enterprise programming languages are OO based, most models will be OO in nature. After all, OO is based on a modelling paradigm. The concepts of the model will be represented as classes and interfaces, the responsibilities as class members.

**Speaking the Language**

Let’s now look at another bedrock principle of domain-driven design. To recap: we want to build a domain model that captures the problem domain of the system being built, and we’re going to express that understanding in code / software artifacts. To help us do that, DDD advocates that the domain experts and developers consciously communicate using the concepts within the model. So the domain experts don’t describe a new user story in terms of a field on a screen or a menu item, they talk about the underlying property or behaviour that’s required on a domain object. Similarly the developers don’t talk about new instance variables of a class or columns in a database table.

Doing this rigorously we get to develop a **ubiquitous language**. If an idea can’t easily be expressed then it indicates a concept that’s missing from the domain model and the team work together to figure out what that missing concept is. Once this has been established then the new field on the screen or column in the database table follows on from that.

Like much of DDD, this idea of developing a ubiquitous language isn’t really a new idea: the XPers call it a "system of names", and DBAs for years have put together data dictionaries. But ubiquitous language *is* an evocative term, and something that can be sold to business and technical people alike. It also makes a lot of sense now that "whole team" agile practices are becoming mainstream.

**Models and Contexts …**

Whenever we discuss a model it's always within some context. This context can usually be inferred from the set of end-users that use the system. So we have a front-office trading system deployed to traders, or a point-of-sale system used by cashiers in a supermarket. These users relate to the concepts of the model in a particular way, and the terminology of the model makes sense to these users but not necessarily to anyone else outside that context. DDD calls this the ***bounded context** (BC)*. Every domain model lives in precisely one BC, and a BC contains precisely one domain model.

I must admit when I first read about BCs I couldn't see the point: if BCs are isomorphic to domain models, why introduce a new term? If it were only end-users that interacted with BCs, then perhaps there wouldn't be any need for this term. However, different systems (BCs) also interact with each other, sending files, passing messages, invoking APIs, etc. If we know there are two BCs interacting with each other, then we know we must take care to translate between the concepts in one domain and those of the other.

Putting an explicit boundary around a model also means we can start discussing relationships between these BCs. In fact, DDD identifies a whole set of relationships between BCs, so that we can rationalize as to what we should do when we need to link our different BCs together:

- **published language**: the interacting BCs agree on a common a language (for example a bunch of XML schemas over an enterprise service bus) by which they can interact with each other;
- **open host service**: a BC specifies a protocol (for example a RESTful web service) by which any other BC can use its services;
- **shared kernel**: two BCs use a common kernel of code (for example a library) as a common lingua-franca, but otherwise do their other stuff in their own specific way;
- **customer/supplier**: one BC uses the services of another and is a stakeholder (customer) of that other BC. As such it can influence the services provided by that BC;
- **conformist**: one BC uses the services of another but is not a stakeholder to that other BC. As such it uses "as-is" (conforms to) the protocols or APIs provided by that BC;
- **anti-corruption layer**: one BC uses the services of another and is not a stakeholder, but aims to minimize impact from changes in the BC it depends on by introducing a set of adapters – an anti-corruption layer.

You can see as we go down this list that the level of co-operation between the two BCs gradually reduces (see Figure 2). With a **published language** we start off with the BCs establishing a common standard by which they can interact; neither owns this language, rather it is owned by the enterprise in which they reside (it might even be an industry standard). With **open host** we're still doing pretty well; the BC provides its functionality as a runtime service for any other BC to invoke but will (presumably) maintain backwards compatibility as the service evolves.

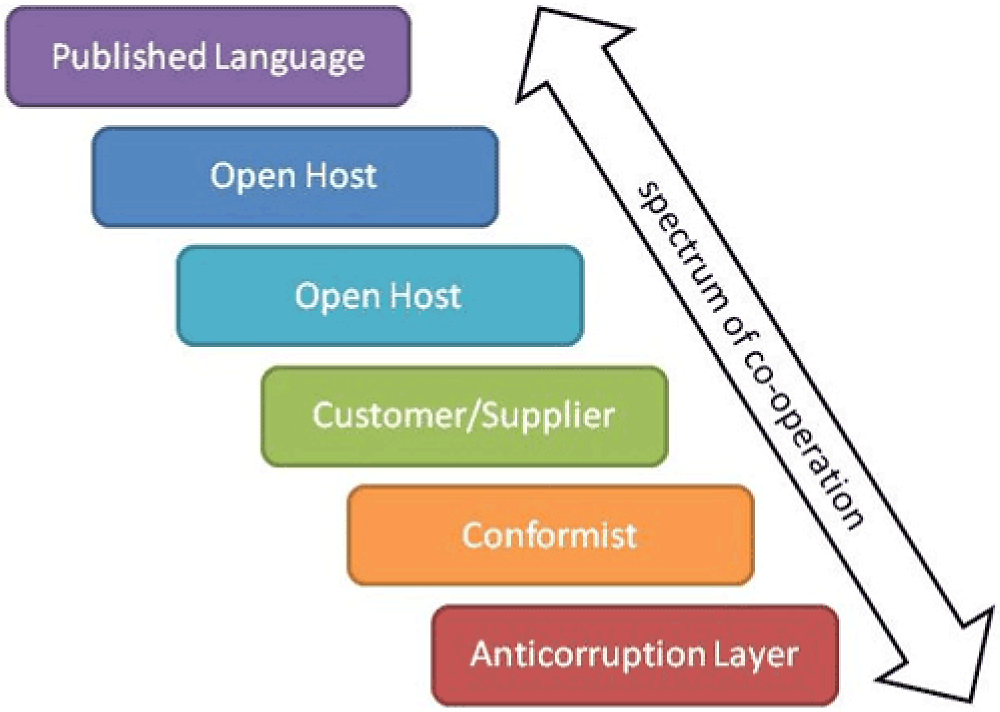


Figure 2: Spectrum of Bounded Context Relationships

However, by the time we get down to **conformist** we are just living with our lot; one BC is clearly subservient to the other. If we had to integrate with the general ledger system, purchased for megabucks, that might well be the situation we'd live in. And if we use an **anti-corruption layer** then we're generally integrating with a legacy system, but introduce an extra layer to isolate ourselves as best we can from it. That costs money to implement, of course, but it reduces the dependency risk. An anti-corruption layer is also lot cheaper than re-implementing that legacy system, something that at best would distract our attention from the core domain, and at worst would end in failure.

DDD suggests that we draw up a **context map** to identify our BCs and those on which we depend or are depended, identifying the nature of these dependencies. Figure 3 shows such a context map for a system I've been working on for the last 5 years or so.

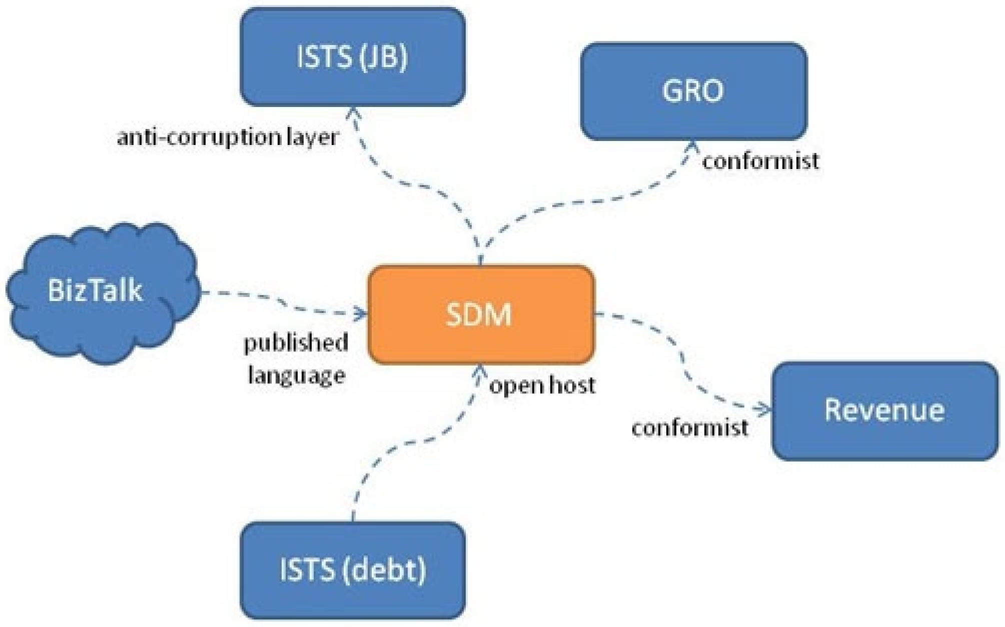


Figure 3: Context Mapping Example

All this talk about context maps and BCs is sometimes called **strategic DDD**, and for good reason. After all, figuring out the relationship between BCs is all pretty political when you think about it: which upstream systems will my system depend on, is it easy for me to integrate with them, do I have leverage over them, do I trust them? And the same holds true downstream: which systems will be using my services, how do I expose my functionality as services, will they have leverage over me? Misunderstand this and your application could easily be a failure.

**Layers and Hexagons**

Let's now turn inwards and consider the architecture of our own BC (system). Fundamentally DDD is only really concerned about the domain layer and it doesn't, actually, have a whole lot to say about the other layers: presentation, application or infrastructure (or persistence layer). But it does expect that they exist. This is the **layered architecture** pattern (Figure 4).

Presentation Layer
Application Layer
Domain Layer
Infrastructure Layer

Figure 4: Layered Architecture

We've been building multi-layer systems for years, of course, but that doesn't mean we're necessarily that good at it. Indeed, some of the dominant technologies of the past – yes, EJB 2, I'm looking at you! – have been positively harmful to the idea that a domain model can exist as a meaningful layer. All the business logic seems to seep into the application layer or (even worse) presentation layer, leaving a set of anaemic domain classes [3] as an empty husk of data holders. This is not what DDD is about.

So, to be absolutely clear, there shouldn't be any domain logic in the application layer. Instead, the application layer takes responsibility for such things as transaction management and security. In some architectures it may also take on the responsibility of ensuring that domain objects retrieved from the infrastructure/persistence layer are properly initialized before being interacted with (though I prefer it that the infrastructure layer does this instead).

Where the presentation layer runs in a separate memory space then the application layer also acts as a mediator between the presentation layer and domain layer. The presentation layer generally deals with serializable representations of a domain object or domain objects (data transfer objects, or DTOs), typically one per "view". If these are modified then the presentation layer sends back any changes to the application layer, which in turn determines the domain objects that have been modified, loads them from the persistence layer, and then forwards on the changes to those domain objects.

One downside of the layered architecture is that it suggests a linear stacking of dependencies, from the presentation layer all the way down to the infrastructure layer. However, we may want to support different implementations within both the presentation and infrastructure layer. That's certainly the case if (as I presume we are!) we want to test our application:

- for example, tools such as FitNesse [4] allow us to verify the behaviour of our system from an end-users' perspective. But these tools don't generally go through the presentation layer, instead they go directly to the next layer back, the application layer. So in a sense FitNesse acts as an alternative viewer.
- similarly, we may well have more than one persistence implementation. Our production implementation probably uses RDBMS or similar technology, but for testing and prototyping we may have a lightweight implementation (perhaps even in-memory) so we can mock out persistence.

We might also want to distinguish between interactions between the layers that are "internal" and "external", where by internal I mean an interaction where both layers are wholly within our system (or BC), while an external interaction goes across BCs.

So rather than consider our application as a set of layers, an alternative is to view it as a hexagon [5], as shown in Figure 5. The viewers used by our end-users, as well as FitNesse testing, use an internal client API (or port), while calls coming in from other BCs (eg RESTful for an open host interaction, or an invocation from an ESB adapter for a published language interaction) hit an external client port. For the back-end infrastructure layer we can see a persistence port for alternative object store implementations, and in addition the objects in our domain layer can call out to other BCs through an external services port.

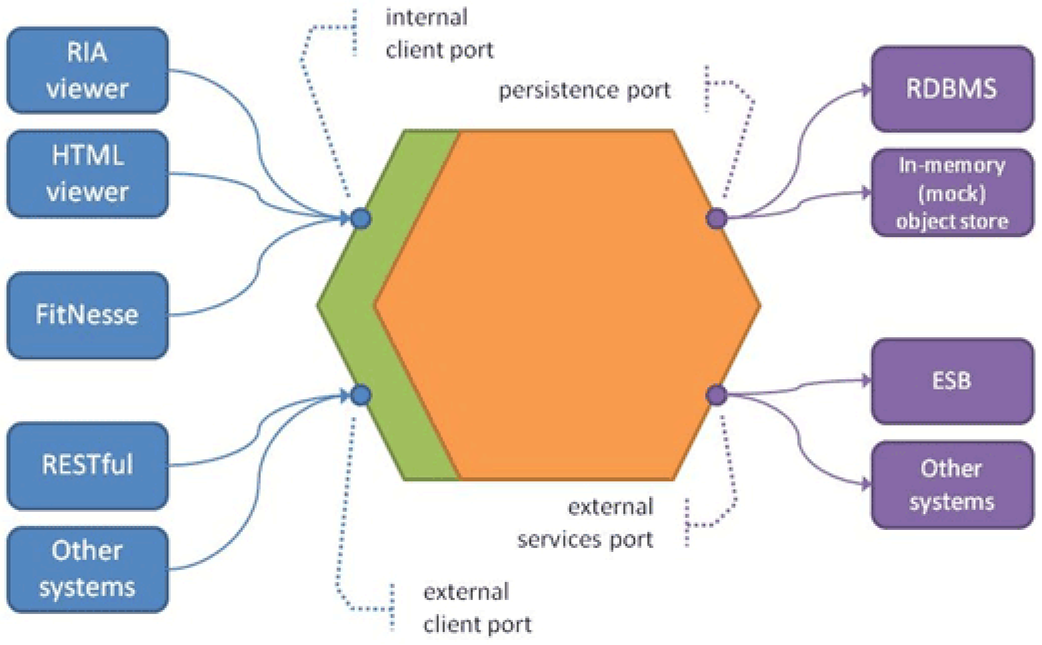


Figure 5: Hexagonal Architecture

But enough of this large-scale stuff; let’s get down to what DDD looks like at the coal-face.

**Building Blocks**

As we’ve already noted, most DDD systems will likely use an OO paradigm. As such many of the building blocks for our domain objects may well be familiar, such as **entities, value objects** and **modules**. For example, if you are a Java programmer then it’s safe enough to think of a DDD entity as basically the same thing as a JPA entity (annotated with @Entity); value objects are things like strings, numbers and dates; and a module is a package.

However, DDD tends to place rather more emphasis on **value objects** than you might be used to. So, yes, you can use a String to hold the value of a *Customer*’s *givenName* property, for example, and that would probably be reasonable. But what about an amount of money, for example the price of a *Product*? We could use an int or a double, but (even ignoring possible rounding errors) what would 1 or 1.0 mean? $1? €1? ¥1? 1 cent, even? Instead, we should introduce a *Money* value type, which encapsulates the *Currency* and any rounding rules (which will be specific to the *Currency*).

Moreover, value objects should be immutable, and should provide a set of side-effect free functions to manipulate them. We should be able to write:

```
Money m1 = new Money("GBP", 10);
Money m2 = new Money("GBP", 20);
Money m3 = m1.add(m2);
```

Adding m2 to m1 does not alter m1, instead it returns a new *Money* object (referenced by m3) which represents the two amounts of *Money* added together.

Values should also have value semantics, which means (in Java and C# for example) that they implement *equals()* and *hashCode()*. They are also often serializable, either into a bytestream or perhaps to and from a *String* format. This is useful when we need to persist them.

Another case where value objects are common is as identifiers. So a (US) *SocialSecurityNumber* would be a good example, as would a *RegistrationNumber* for a *Vehicle*. So would a *URL*. Because we have overridden *equals()* and *hashCode()*, all of these could then safely be used as keys in a hash map.

Introducing value objects not only expands our ubiquitous language, it also means we can push behaviour onto the values themselves. So if we decided that *Money* can never contain negative values, we can implement this check right inside *Money*, rather than everywhere that a *Money* is used. If a *SocialSecurityNumber* had a checksum digit (which is the case in some countries) then the verification of that checksum could be in the value object. And we could ask a *URL* to validate its format, return its scheme (for example http), or perhaps determine a resource location relative to some other *URL*.

Our other two building blocks probably need less explanation. **Entities** are typically persisted, typically mutable and (as such) tend to have a lifetime of state changes. In many architectures an entity will be persisted as row in a database table. **Modules** (packages or namespaces) meanwhile are key to ensuring that the domain model remains decoupled, and does not descend into one big ball of mud [6]. In his book Evans talks about **conceptual contours**, an elegant phrase to describe how to separate out the main areas of concern of the domain. Modules are the main way in which this separation is realized, along with interfaces to ensure that module dependencies are strictly acyclic. We use techniques such as Uncle "Bob" Martin’s dependency inversion principle [7] to ensure that the dependencies are strictly one-way.

Entities, values and modules are the core building blocks, but DDD also has some further building blocks that will be less familiar. Let’s look at these now.

## Aggregates and Aggregate Roots

If you are versed in UML then you’ll remember that it allows us to model an association between two objects either as a simple association, as an aggregation, or using composition. An **aggregate root** (sometimes abbreviated to AR) is an entity that composes other entities (as well as its own values) by composition. That is, aggregated entities are referenced only by the root (perhaps transitively), and may not be (permanently) referenced by any objects outside the aggregate. Put another way, if an entity has a reference to another entity, then the referenced entity must either be within the same aggregate, or be the root of some other aggregate.

Many entities are aggregate roots and contain no other entities. This is especially true for entities that are immutable (equivalent to reference or static data in a database). Examples might include *Country*, *VehicleModel*, *TaxRate*, *Category*, *BookTitle* and so on.

However, more complex mutable (transactional) entities do benefit when modelled as aggregates, primarily by reducing the conceptual overhead. Rather than having to think about every entity we can think only of the aggregate roots; aggregated entities are merely the "inner workings" of the aggregate. They also simplify the interactions between entities; we follow the rule that (persisted) references may only be to an aggregate's root, not to any other entities within the aggregate.

Another DDD principle is that an aggregate root is responsible for ensuring that the aggregated entities are always in a valid state. For example, an *Order* (root) might contain a collection of *OrderItem*s (aggregated). There could be a rule that any *OrderItem* cannot be updated once the *Order* has been shipped. Or, if two *OrderItem*s refer to the same *Product* and with the same shipping requirements, then they are merged into the same *OrderItem*. Or, an *Order*'s derived *totalPrice* attribute should be the sum of the prices of the *OrderItem*s. Maintaining these invariants is the responsibility of the root.

However … it is only feasible for an aggregate root to maintain invariants between objects entirely within the aggregate. The *Product* referenced by an *OrderItem* would almost certainly not be in the AR, because there are other use cases which need to interact with *Product* independently of whether there is an *Order* for it. So, if there were a rule that an *Order* cannot be placed against a discontinued *Product*, then the *Order* would need to deal with this somehow. In practical terms this generally means using isolation level 2 or 3 to "lock" the *Product* while the *Order* is updated transactionally. Alternatively, an out-of-band process can be used to reconcile any breakage of cross-aggregate invariants.

Stepping back a little before we move on, we can see that we have a spectrum of granularity:

value < entity < aggregate < module < bounded context

Let's now carry on looking at some further DDD building blocks.

**Repositories, Factories and Services**

In enterprise applications entities are typically persisted, with values representing the state of those entities. But how do we get hold of an entity from the persistence store in the first place?

A **repository** is an abstraction over the persistence store, returning entities – or more precisely aggregate roots – meeting some criteria. For example, a customer repository would return *Customer* aggregate root entities, and an order repository would return *Order*s (along with their *OrderItem*s). Typically there is one repository per aggregate root.

Because we generally want to support multiple implementations of the persistence store, a repository typically consists of an interface (eg *CustomerRepository*) with different implementations for the different persistence store implementations (eg *CustomerRepositoryHibernate*, or *CustomerRepositoryInMemory*). Because this interface returns entities (part of the domain layer), the interface itself is also part of the domain layer. The implementations of the interface (coupled as they are to some specific persistence implementation) are part of the infrastructure layer.

Often the criteria we are searching for is implicit in the method name called. So *CustomerRepository* might provide a *findByLastName(String)* method to return *Customer* entities with the specified last name. Or we could have an *OrderRepository* to return *Order*s, with a *findByOrderNum(OrderNum)* returning the *Order* matching the *OrderNum* (note the use of a value type here, by the way!).

More elaborate designs wrap the criteria into a query or a specification, something like *findBy(Query<T>)*, where *Query* holds an abstract syntax tree describing the criteria. The different implementations then unwrap the *Query* to determine how to locate the entities meeting the criteria in their own specific way.

That said, if you are a .NET developer then one technology that warrants mention here is LINQ [8]. Because LINQ is itself pluggable, it is often the case that we can write a single implementation of the repository using LINQ. What then varies is not the repository implementation but the way in which we configure LINQ to obtain its data sources (eg against Entity Framework or against an in-memory objectstore).

A variation on using specific repository interfaces per aggregate root is to use generic repositories, such as *Repository<Customer>*. This provides a common set of methods, such as *findById(int)* for every entity. This works well when using *Query<T>* (eg *Query<Customer>*) objects to specify the criteria. For the Java platform there are also frameworks such as Hades [9] that allow a mix-and-match approach (start with generic implementations, and then add in custom interfaces as and when are needed).

Repositories aren't the only way to bring in objects from the persistence layer. If using an object-relational mapping (ORM) tool such as Hibernate we can navigate references between entities, allowing us to transparently walk the graph. As a rule of thumb references to the aggregate roots of other entities should be lazy loaded, while aggregated entities within an aggregate should be eagerly loaded. But as ever with ORMs, expect to do some tuning in order to obtain suitable performance characteristics for your most critical use cases.

In most designs repositories are also used to save new instances, and to update or delete existing instances. If the underlying persistence technology supports it then these may well live on a generic repository, though … from a method signature point of view there's nothing really to distinguish saving a new *Customer* from saving a new *Order*.

One final point on this… it's quite rare to create new aggregate roots directly. Instead they tend to be created by other aggregate roots. An *Order* is a good example: it would probably be created by invoking an action by the *Customer*.

Which neatly brings us onto:

## Factories

If we ask an *Order* to create an *OrderItem*, then (because after all the *OrderItem* is part of its aggregate) it's reasonable for the *Order* to know the concrete class of *OrderItem* to instantiate. In fact, it's reasonable for an entity to know the concrete class of any entity within the same module (namespace or package) that it needs to instantiate.

Suppose though that the *Customer* creates an *Order*, using the *Customer*'s *placeOrder* action (see Figure 6). If the *Customer* knows the concrete class of *Order*, that means that the *customer* module is dependent on the *order* module. If the *Order* has a back-reference to the *Customer* then we will get a cyclic dependency between the two modules.

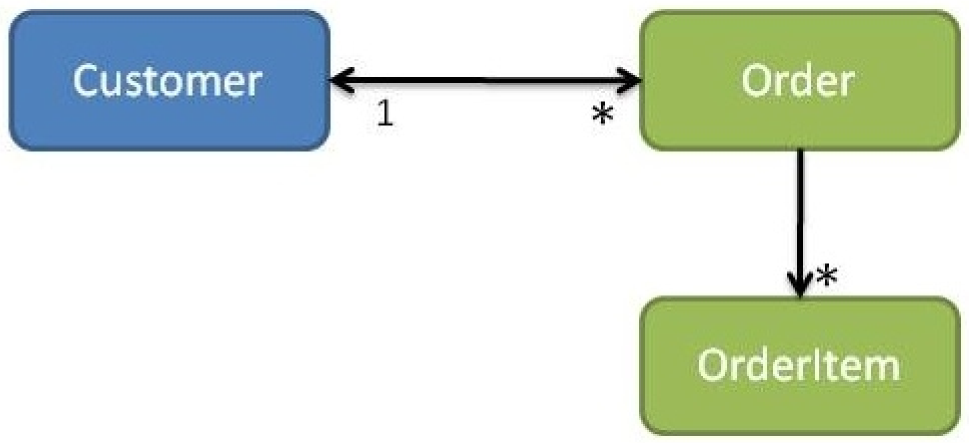


Figure 6: Customers and Orders (cyclic dependencies)

As already mentioned, we can use the dependency inversion principle to resolve this sort of thing: to remove the dependency from *order* -> *customer* module we would introduce an *OrderOwner* interface, make *Order* reference an *OrderOwner*, and make *Customer* implement *OrderOwner* (see Figure 7).

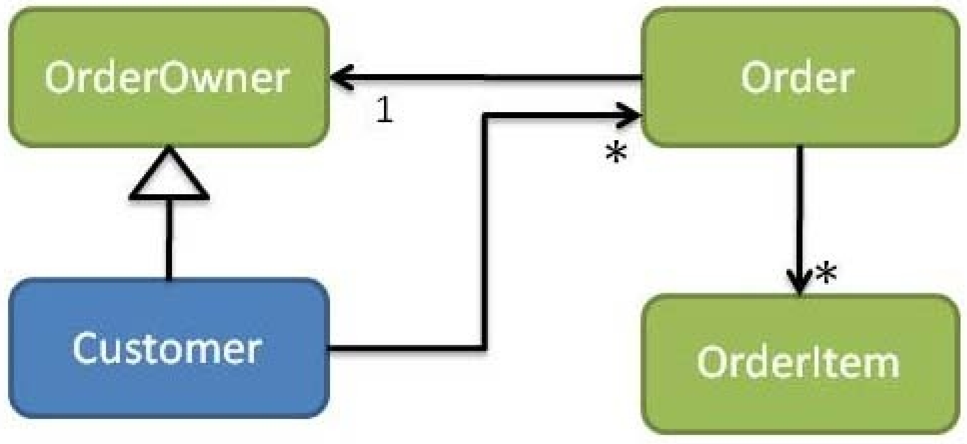


Figure 7: Customers and Orders (customer depends on order)

What about the other way, though: if we wanted *order* -> *customer*? In this case there would need to be an interface representing *Order* in the *customer* module (this being the return type of *Customer*'s *placeOrder* action). The *order* module would then provide implementations of *Order*. Since *customer* cannot depend on *order*, it instead must define an *OrderFactory* interface. The *order* module then provides an implementation of *OrderFactory* in turn (see Figure 8).

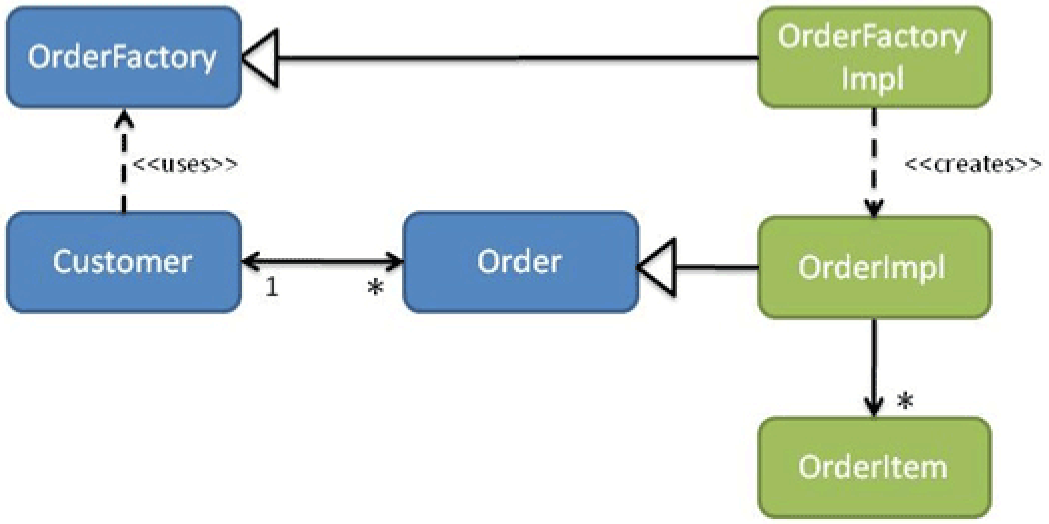


Figure 8: Customers and Orders (order depends on customer)

There might also be a corresponding repository interface. For example, if a *Customer* could have many thousands of *Order*s then we might remove its *orders* collection. Instead the Customer would use an *OrderRepository* to locate (a subset of) its *Order*s as required. Alternatively (as some prefer) you can avoid the explicit dependency from an entity to a repository by moving the call to the repository into a higher layer of the application architecture, such as a **domain service** or perhaps an **application service**.

Indeed, services are the next topic we need to explore.

**Domain services, Infrastructure services and Application services**

A **domain service** is one which is defined within the domain layer, though the implementation may be part of the infrastructure layer. A repository is a domain service whose implementation is indeed in the infrastructure layer, while a factory is also a domain service whose implementation is generally within the domain layer. In particular both repositories and factories are defined within the appropriate module: *CustomerRepository* is in the *customer* module, and so on.

More generally though a domain service is any business logic that does not easily live within an entity. Evans suggests a transfer service between two bank accounts, though I'm not sure that's the best example (I would model a *Transfer* itself as an entity). But another variety of domain service is one that acts as a proxy to some other bounded context. For example, we might want to integrate with a General Ledger system that exposes an open host service. We can define a service that exposes the functionality that we need, so that our application can post entries to the general ledger. These services sometimes define their own entities which may be persisted; these entities in effect shadow the salient information held remotely in the other BC.

We can also get services that are more technical in nature, for example sending out an email or SMS text message, or converting a *Correspondence* entity into a PDF, or stamping the generated PDF with a barcode. Again the interface is defined in the domain layer, but the implementation is very definitely in the infrastructure layer. Because the interface for these very technical services is often defined in terms of simple value types (not entities), I tend to use the term **infrastructure service** rather than domain service. But you could think of them if you want as a bridge over to an "email" BC or an "SMS" BC if you wanted to.

While a domain service can both call or be called by a domain entity, an ***application service*** sits above the domain layer so cannot be called by entities within the domain layer, only the other way around. Put another way, the application layer (of our layered architecture) can be thought of as a set of (stateless) application services.

As already discussed, application services usually handle cross-cutting concerns such as transactions and security. They may also mediate with the presentation layer, by: unmarshalling the inbound request; using a domain service (repository or factory) to obtain a reference to the aggregate root being interacted with; invoking the appropriate operation on that aggregate root; and marshalling the results back to the presentation layer.

I should also point out that in some architectures application services call infrastructure services. So, rather than an entity call a *PdfGenerationService* to convert itself to a PDF, the application service may call the *PdfGenerationService* directly, passing information that it has extracted from the entity. This isn't my particular preference, but it is a common design. I'll talk about this more shortly.

Okay, that completes our overview of the main DDD patterns. There's plenty more in Evans 500+page book – and it's well worth the read – but what I'd like to do next is highlight some areas where people seem to struggle to apply DDD.

## Problems and Hurdles

### Enforcing a layered architecture

Here's the first thing: it can be difficult to strictly enforce the architectural layering. In particular, the seeping of business logic from the domain layer into the application layer can be particularly insidious.

I've already singled out Java's EJB2 as a culprit here, but poor implementations of the model-view-controller pattern can also cause this to happen. What happens is the controller (= application layer), takes on far too much responsibility, leaving the model (= domain layer) to become anaemic. In fact, there are newer web frameworks out there (in the Java world, Wicket [10] is one up-and-coming example) that explicitly avoid the MVC pattern for this sort of reason.

### Presentation layer obscuring the domain layer

Another issue is trying to develop a ubiquitous language. It's natural for domain experts to talk in terms of screens, because, after all, that is all that they can see of the system. Asking them to look behind the screens and express their problem in terms of the domain concepts can be very difficult.

The presentation layer itself can also be problematic, because a custom presentation layer may not accurately reflect (could distort) the underlying domain concepts and so compromises our ubiquitous language. Even if that isn't the case, there's also just the time it takes to put together a user interface.

Using agile terminology, the reduced velocity means less progress is made per iteration, and so fewer insights are made into the domain overall.

**The implementation of the repository pattern**

On a more technical note, newbies also sometimes seem to get confused separating out the interface of a repository (in the domain layer) from its implementation (in the infrastructure layer). I'm not exactly sure why this is the case: it's a pretty simple OO pattern, after all. I think it might be because Evans book just doesn't go down to this level of detail, and that leaves some people high and dry. But it's also possibly because the idea of substituting out the persistence implementation (as per the hexagonal architecture) is not that widespread, giving rise to systems where persistence implementation has bled into the domain layer.

**The implementation of the service dependencies**

Another technical concern – and where there can be disagreement between DDD practitioners – is in terms of the relationship between entities and domain/infrastructure services (including repositories and factories). Some take the view that entities should not depend on domain services at all, but if that's the case then it falls to the outer application services to interact with the domain services and pass the results into the domain entities. To my way of thinking this moves us towards having an anaemic domain model.

A slightly softer viewpoint is that entities can depend on domain services, but the application service should pass them in as needed, for example as an argument to an operation. I'm not a fan of this either: to me it is exposing implementation details out to the application layer ("this entity needs such-and-such a service in order to fulfil this operation"). But many practitioners are happy with this approach.

My own preferred option is to inject services into entities, using dependency injection. Entities can declare their dependencies and then the infrastructure layer (eg Hibernate, Spring or some other framework) can inject the services into the entities:

```
public class Customer {
…
private OrderFactory orderFactory;
public void setOrderFactory(OrderFactory orderFactory) {
this.orderFactory = orderFactory;
}
…
public Order placeOrder( … ) {
Order order = orderFactory.createOrder();
…
return order;
}
}
```

One alternative is to use the service locator pattern. All services are registered, for example into JNDI, and then each domain object looks up the services it requires. To my mind this introduces a dependency on the runtime environment. However, compared to dependency injection it has lower memory requirements on entities, and that could be a deciding factor.

**Inappropriate modularity**

As we’ve already identified, DDD distinguishes several different levels of granularity over and above entities, namely aggregates, modules and BCs. Getting the right levels of modularization right takes some practice. Just as an RDBMS schema may be denormalized, so too can a system not have modularity (becomes a big ball of mud). But an overnormalized RDBMS schema – where a single entity is broken out over multiple tables – can also be harmful, and so too can an overmodularized system, because it then becomes difficult to understand how the system works as a whole.

Let’s first consider modules and BCs. A module, remember, is akin to a Java package or .NET namespace. We want the dependency between two modules to be acyclic, for sure, but if we do decide that (say) *customer* depends on *order* then there’s nothing in particular extra we need to do: *Customer* can simply import the *Order* package/namespace and uses its interfaces and classes as needed.

If we put the *customer* and *order* into separate BCs, however, then we have substantially more work to do, because we must map the concepts in the *customer* BC to those of the *order* BC. In practice this also means having representations of the *order* entities in the *customer* BC (per the general ledger example given earlier), as well as the mechanics of actually collaborating via message bus or something else. Remember: the reason for having two BCs is when there are different end-users and/or stakeholders, and we can’t guarantee that the related concepts in the different BCs will evolve in the same direction.

Another area where there can be confusion is in distinguishing entities from aggregates. Every aggregate has an entity acting as its aggregate root, and for lots and lots of entities the aggregate will consist of just this entity (the "trivial" case, as mathematicians would say). But I’ve seen developers fall into thinking that the whole world must reside within a single aggregate. So, for example, *Order* contains *OrderItem*s (so far so good) which reference *Product*s, and so the developer concludes that *Product*s are also in the aggregate (no!) Even worse, a developer will observe that *Customer* have *Order*s, and so think this means we must have mega-aggregate of *Customer* / *Order* / *OrderItem* / *Product* (no, no, no!). The point is that "*Customer* have *Order*s" does not mean imply aggregation; *Customer*, *Order* and *Product* are all aggregate roots.

In practical terms, a typical module (and this is very rough and ready) might contain a half-dozen aggregates, each of which might contain between one entity and several. Of these half-dozen, a good number may well be immutable "reference data" classes. Remember also that the reason we modularize is so that we can understand a single thing (at a certain level of granularity). So do remember that the typical person can only keep in their head between 5 and 9 things at a time [11].

**Getting Started**

As I said at the outset, you might well have encountered many of the ideas in DDD before. Indeed, every Smalltalker I’ve spoken with (I’m not one, I’m afraid to say) seems glad to be able to go back to a domain-driven approach after the wilderness years of EJB2 et al.

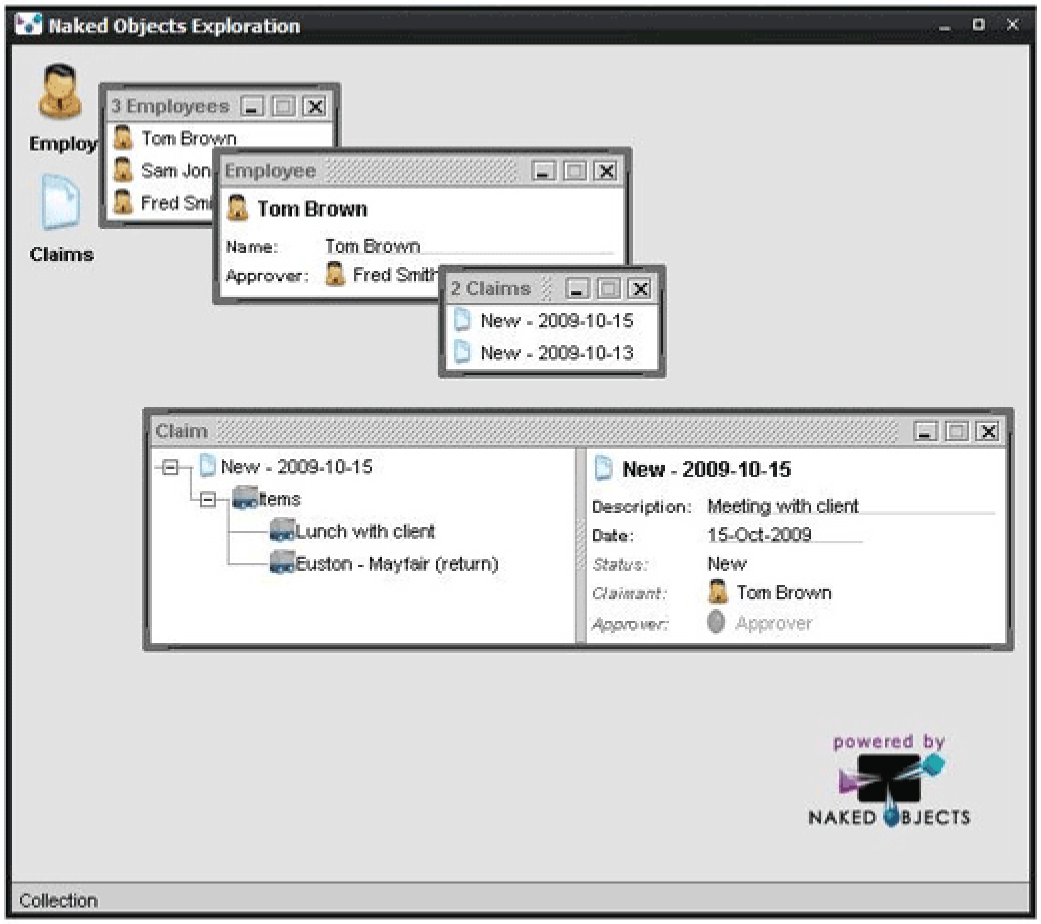
On the other hand, what if this stuff is new? With so many different ways to trip up, is there any way to reliably get started with DDD?

If you look around the Java landscape (it's not as bad for .NET) there are literally hundreds of frameworks for building web apps (JSP, Struts, JSF, Spring MVC, Seam, Wicket, Tapestry, etc). And there are scores of frameworks out there targeting the infrastructure layer, both from a persistence perspective (JDO, JPA, Hibernate, iBatis, TopLink, JCloud and so on) or other concerns (RestEasy, Camel, ServiceMix, Mule etc). But there are very few frameworks or tools to help with what DDD says is the most important layer, the domain layer.

Since 2002 I've been involved in with (and these days am a committer to) a project called Naked Objects, open source on Java [12] and commercial on .NET [13]. Although Naked Objects wasn't explicitly started with domain-driven design in mind – indeed it predates Evans' book – it holds by a very similar set of principles to DDD. It also makes it easy to overcome the hurdles identified earlier.

You can think of Naked Objects analogously to an ORM such as Hibernate. Whereas ORMs build a metamodel of the domain objects and use this to automatically persist the domain objects to an RDBMS, Naked Objects builds a metamodel and uses this to automatically render those domain objects in an object-oriented user interface.

Out-of-the-box Naked Objects supports two user interfaces, a rich client viewer (see Figure 9), and a HTML viewer (see Figure 10). These are both fully functional applications that require the developer to write only the domain layer (entities, values, repositories, factories, services) to run.



Tom Brown
Sam Jon
Fred Smi

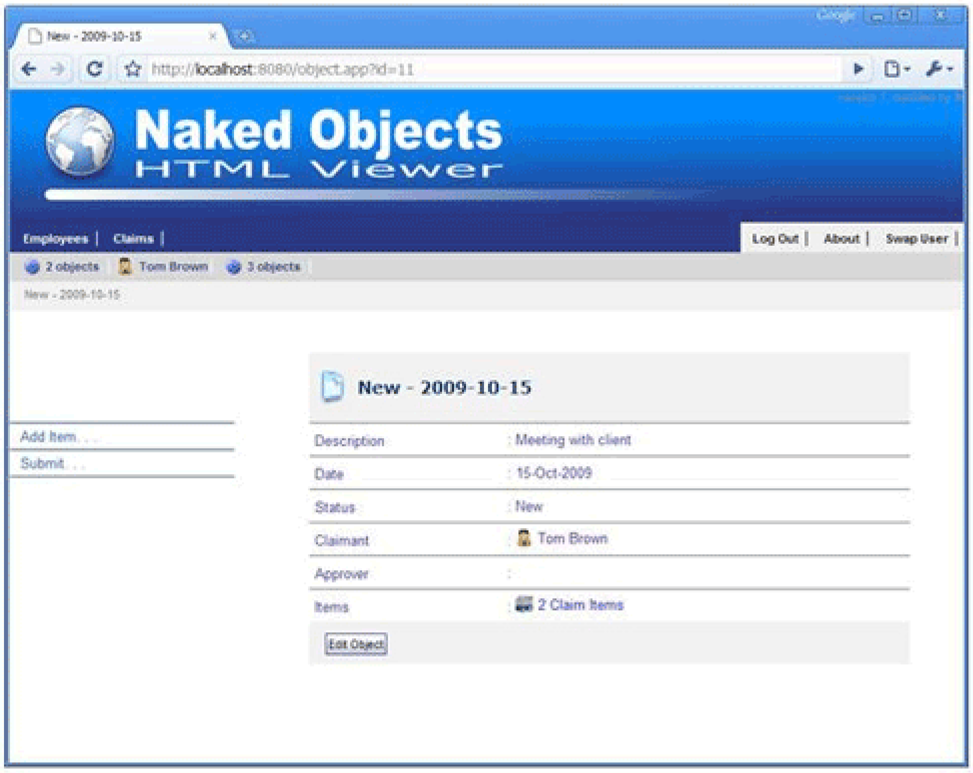
Tom Brown
Name: Tom Brown
Approver: Fred Smith

New - 2009-10-15
New - 2009-10-13

New - 2009-10-15
Items
Lunch with client
Euston - Mayfair (return)

New - 2009-10-15
Description: Meeting with client
Date: 15-Oct-2009
Status: New
Claimant: Tom Brown
Approver: Approver

Figure 9: Naked Objects Drag-n-Drop Viewer



New - 2009-10-15

|  |  |
|---|---|
| Description | : Meeting with client |
| Date | : 15-Oct-2009 |
| Status | : New |
| Claimant | : Tom Brown |
| Approver | : |
| Items | : 2 Claim Items |

Figure 10: Naked Objects HTML Viewer

Let’s take a look at the (Java) code for the *Claim* class (shown in the screenshots). First off, the classes are basically pojos, though we normally inherit from the convenience class *AbstractDomainObject* just to factor out the injection of a generic repository and provide some helper methods:

```
public class Claim extends AbstractDomainObject {
...
}
Next, we have some value properties:
// {{ Description
private String description;
@MemberOrder(sequence = "1")
public String getDescription() { return description; }
public void setDescription(String d) { description = d; }
// }}

// {{ Date
private Date date;
@MemberOrder(sequence="2")
public Date getDate() { return date; }
public void setDate(Date d) { date = d; }
// }}

// {{ Status
private String status;
@Disabled
@MemberOrder(sequence = "3")
public String getStatus() { return status; }
public void setStatus(String s) { status = s; }
// }}
```

These are simple getter/setters, with return types of String, dates, integers and so on (though Naked Objects supports custom value types too). Next, we have some reference properties:

```
// {{ Claimant
private Claimant claimant;
@Disabled
@MemberOrder(sequence = "4")
public Claimant getClaimant() { return claimant; }
public void setClaimant(Claimant c) { claimant = c; }
// }}

// {{ Approver
private Approver approver;
@Disabled
@MemberOrder(sequence = "5")
public Approver getApprover() { return approver; }
public void setApprover(Approver a) { approver = a; }
// }}
```

Here our *Claim* entity references other entities. In fact, *Claimant* and *Approver* are interfaces, so this allows us to decouple our domain model into modules as discussed earlier on.

Entities can also have collections of entities. In our case *Claim* has a collection of *ClaimItem*s:

```
// {{ Items
private List<ClaimItem> items = new
ArrayList<ClaimItem>();
@MemberOrder(sequence = "6")
public List<ClaimItem> getItems() { return items; }
public void addToItems(ClaimItem item) {
items.add(item);
}
// }}
```

And we also have (what Naked Objects calls) actions, namely *submit* and *addItem*: these are all public methods that don't represent properties and collections:

```
// {{ action: addItem
public void addItem(
@Named("Days since")
int days,
@Named("Amount")
double amount,
@Named("Description")
String description) {
ClaimItem claimItem = newTransientInstance(ClaimItem.class);
Date date = new Date();
date = date.add(0,0, days);
claimItem.setDateIncurred(date);
claimItem.setDescription(description);
claimItem.setAmount(new Money(amount, "USD"));
persist(claimItem);
addToItems(claimItem);
}
public String disableAddItem() {
```

```
return "Submitted".equals(getStatus()) ? "Already
submitted" : null;
}
// }}
// {{ action: Submit
public void submit(Approver approver) {
setStatus("Submitted");
setApprover(approver);
}
public String disableSubmit() {
return getStatus().equals("New")?
null : "Claim has already been submitted";
}
public Object[] defaultSubmit() {
return new Object[] { getClaimant().getApprover() };
}

// }}
```

These actions are automatically rendered in the Naked Objects viewers as menu items or links. And it's the presence of these actions that mean that Naked Objects apps are not just CRUD-style applications.

Finally, there are supporting methods to display a label (or title) and to hook into the persistence lifecycle:

```
// {{ Title
public String title() {
return getStatus() + " - " + getDate();
}
// }}

// {{ Lifecycle
public void created() {
status = "New";
date = new Date();
}
// }}
```

Earlier I described Naked Objects domain objects as pojos, but you'll have noticed that we use annotations (such as *@Disabled*) along with imperative helper methods (such as *disableSubmit()*) to enforce business constraints. The Naked Objects viewers honour these semantics by querying the metamodel built at startup. If you don't like these programming conventions, it is possible to change them.

A typical Naked Objects application consists of a set of domain classes such as the *Claim* class above, along with interfaces and implementations for repository, factory and domain/infrastructure services. In particular, there is no presentation layer or application layer code. So how does Naked Objects help with some of the hurdles we've already identified? Well:

- *Enforcing a layered architecture*: because the only code we write are the domain objects, there's no way that domain logic can seep out into other layers. Indeed, one of the original motivations for Naked Objects was to help develop behaviourally complete objects
- *Presentation layer obscuring the domain layer*: because the presentation layer is a direct

reflection of the domain objects, the whole team can rapidly deepen their understanding of the domain model. By default Naked Objects takes the class names and method names straight from the code, so there is a strong incentive to get the naming right in the ubiquitous language. In this way Naked Objects also supports DDD's model-driven design principle

- *The implementation of the repository pattern*: the icons / links that you can see in the screenshots are actually repositories: an *EmployeeRepository* and a *ClaimRepository*. Naked Objects supports pluggable object stores, and normally in prototyping we use an implementation against an in-memory object store. As we move towards production, we then write an implementation that hits the database.
- *The implementation of the service dependencies*: Naked Objects automatically injects service dependencies into every domain object. This is done when the object is retrieved from the object store, or when the object is first created (see *newTransientInstance()*, above). In fact, all that such helper methods do is delegate to a generic repository/factory provided by Naked Objects called the *DomainObjectContainer*.
- *Inappropriate modularity*: We can modularize into modules just by using Java packages (or .NET namespaces) in the normal way, and use visualization tools such as Structure101 [14] and NDepend [15] to ensure that there are no cyclic dependencies in our codebase. We can modularize into aggregates by annotating as @Hidden any aggregated objects that represent the inner workings of our visible aggregate root; these then won't appear in the Naked Objects viewers. And we can write domain and infrastructure services to bridge over to other BCs as we need to.

There are plenty of other features that Naked Objects offers: it has an extensible architecture that – in particular – allows other viewers and object stores to be implemented. The next generation of viewers being developed (for example Scimpi [16]) offer more sophisticated customisation capabilities. In addition, it offers multiple deployment options: for example, you can use Naked Objects just for prototyping and then develop your own more tailored presentation layer as you move towards production. It also has integration with tools such as FitNesse [17], and can automatically provide a RESTful interface to your domain objects [18].

## Next Steps

Domain-driven design brings together a set of best-practice patterns for developing complex enterprise applications. Some developers will have been applying these patterns for years, and for these guys DDD is perhaps little more than an affirmation of their existing practices. For others though, applying these patterns can be a real challenge.

Naked Objects provides a framework for both Java and .NET that, by taking care of the other layers, allows the team to focus on the bit that matters, the domain model. By exposing the domain objects directly in the UI, Naked Objects allows the team to very naturally build up an unambiguous ubiquitous language. As the domain layer firms up then the team can develop a more tailored presentation layer if required.

So, where next?

Well, the bible for DDD itself is Eric Evans original book, "Domain-Driven Design" [1], recommended reading for all. The Yahoo newsgroup for DDD [19] is also a pretty good resource. And if you're interested in learning more about Naked Objects, you can search out my book, "Domain Driven Design using Naked Objects" [20], or my blog [21] (NO for Java) or the Naked Objects website [13] (NO for .NET). Happy DDD'ing!

## References

[1] Domain Driven Design Community http://domaindrivendesign.org/

[2] Spring BeanDoc http://spring-beandoc.sourceforge.net/

[3] Anaemic Domain Model, Martin Fowler http://martinfowler.com/bliki/AnemicDomainModel.html

[4] FitNesse http://fitnesse.org

[5] Hexagonal Architecture, Alistair Cockburn http://alistair.cockburn.us/Hexagonal+architecture

[6] Big Ball of Mud,> Brian Foote & Joseph Yoder http://www.laputan.org/mud/

[7] Dependency Inversion Principle, Robert Martin
http://www.objectmentor.com/resources/articles/dip.pdf

[8] LINQ http://msdn.microsoft.com/en-us/netframework/aa904594.aspx

[9] Hades http://hades.synyx.org/

[10] Apache Wicket Web Framework http://wicket.apache.org

[11] Magical Number Seven, ±2 http://en.wikipedia.org/wiki/The_Magical_
Number_Seven,_Plus_or_Minus_Two

[12] Naked Objects for Java http://nakedobjects.org

[13] Naked Objects for .NET http://nakedobjects.net

[14] Structure101 (for Java) http://www.headwaysoftware.com/products/structure101

[15] NDepend (for .NET) http://www.ndepend.com/

[16] Scimpi http://scimpi.org

[17] Tested Objects (FitNesse for Naked Objects) http://testedobjects.sourceforge.net

[18] Restful Objects (REST for Naked Objects) http://restfulobjects.sourceforge.net

[19] Yahoo DDD Newsgroup http://tech.groups.yahoo.com/group/domaindrivendesign/

[20] Domain Driven Design using Naked Objects, Dan Haywood http://pragprog.com/titles/dhnako

[21] Dan Haywood's Blog http://danhaywood.com

## Related Methods & Tools articles

- Domain-Specific Modeling for Full Code Generation
- Mass Customizing Solutions with Software Factories
- Introduction to the Emerging Practice of Software Product Line Development

**More Domain Driven Design Knowledge**

Strategic Design by Eric Evans

Is Domain-Driven Design more than Entities and Repositories?

Use of Domain Driven Design in Enterprise Application Development