AnemicDomainModel

Martin Fowler
25 November 2003

This is one of those anti-patterns that's been around for quite a long time, yet seems to be having a particular spurt at the moment. I was chatting with Eric Evans on this, and we've both noticed they seem to be getting more popular. As great boosters of a proper Domain Model, this is not a good thing.

The basic symptom of an Anemic Domain Model is that at first blush it looks like the real thing. There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have. The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters. Indeed often these models come with design rules that say that you are not to put any domain logic in the the domain objects. Instead there are a set of service objects which capture all the domain logic. These services live on top of the domain model and use the domain model for data.

The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together. The anemic domain model is really just a procedural style design, exactly the kind of thing that object bigots like me (and Eric) have been fighting since our early days in Smalltalk. What's worse, many people think that anemic objects are real objects, and thus completely miss the point of what object-oriented design is all about.

Now object-oriented purism is all very well, but I realize that I need more fundamental arguments against this anemia. In essence the problem with anemic domain models is that they incur all of the costs of a domain model, without yielding any of the benefits. The primary cost is the awkwardness of mapping to a database, which typically results in a whole layer of O/R mapping. This is worthwhile iff you use the powerful OO techniques to organize complex logic. By pulling all the behavior out into services, however, you essentially end up with Transaction Scripts, and thus lose the advantages that the domain model can bring. As I

discussed in P of EAA, Domain Models aren't always the best tool.

It's also worth emphasizing that putting behavior into the domain objects should not contradict the solid approach of using layering to separate domain logic from such things as persistence and presentation responsibilities. The logic that should be in a domain object is domain logic - validations, calculations, business rules - whatever you like to call it. (There are cases when you make an argument for putting data source or presentation logic in a domain object, but that's orthogonal to my view of anemia.)

One source of confusion in all this is that many OO experts do recommend putting a layer of procedural services on top of a domain model, to form a Service Layer. But this isn't an argument to make the domain model void of behavior, indeed service layer advocates use a service layer in conjunction with a behaviorally rich domain model.

Eric Evans's excellent book Domain Driven Design has the following to say about these layers.

> *Application Layer [his name for Service Layer]: Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.*
>
> *Domain Layer (or Model Layer): Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This layer is the heart of business software.*

The key point here is that the Service Layer is thin - all the key logic lies in the domain layer. He reiterates this point in his service pattern:

> *Now, the more common mistake is to give up too easily on fitting the behavior into an appropriate object, gradually slipping toward procedural programming.*

I don't know why this anti-pattern is so common. I suspect it's due to many people who haven't really worked with a proper domain model, particularly if they come from a data background. Some technologies

encourage it; such as J2EE's Entity Beans which is one of the reasons I prefer POJO domain models.

In general, the more behavior you find in the services, the more likely you are to be robbing yourself of the benefits of a domain model. If all your logic is in services, you've robbed yourself blind.