Entity View

Simplify Data Access in Domain-Driven Applications

Tim Wellhausen

kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

v1.0, August 13, 2015

Intent

In a domain-driven application, the Entity View pattern can be used to provide cohesive information about an entity that cannot be retrieved from the entity directly. The pattern is particularly useful if some client requires data about an entity that is stored beyond the entity's boundaries. In that case a service facade aggregates the relevant data and provides it in form of a dedicated object: an entity view.

Introduction

Domain-driven design [1] is an approach with a focus on the business domain. Domain-driven design is often applied in the context of enterprise information systems with complex data models. In such a system, data is typically spread among many places where boundaries between the entities limit their accessibility from each other. In particular, the advent of polyglot persistence [9] has increased the number of database systems where data of a single application is kept. In addition, virtually every enterprise information system requires data from some other systems that can be retrieved by remote service calls only.

Such a complexity comes at a price: the implementation of a single use case may require calls to several services and repositories to gather all data that is needed. For example, a single web page may render information from multiple sources. Or the application may need to provide detailed information about an entity to a full-text index.

A solution to this problem is the creation of objects that aggregate and carry data. In the context of remote interfaces, the Data Transfer Object pattern [5] is an established means for this purpose. The Entity View pattern that is presented here is similar to the Data Transfer Object pattern. The intent of an entity view also is to aggregate and transfer data. However, entity views are an integral part of the domain model and only used within the application. The section on the Entity View pattern itself discusses the distinction between both patterns in detail.

This paper has three parts: the first part gives some background information on domain-driven design that sets the context for the Entity View pattern. The second part introduces the Entity View pattern. And the third part gives an in-depth example how to apply the pattern.

Domain-Driven Design

Domain-driven design [1] is an approach to software development that places focus on the business domain. It emphasizes the importance of a well-designed domain layer that deeply reflects the actual domain and makes it easy to implement domain-related functionality in the system.

The domain model is a model that encapsulates both the data and the behavior of the conceptual domain. In his book "*Domain-driven Design*" [1], Eric Evans defines the building blocks of a domain model, namely, entities, aggregates, repositories and services (among others).

An Entity is a cohesive data structure that has a unique identity. Entities represent the data upon which the system is built. Entities are connected to each other by associations, either directly in form of traversable references in an object graph or indirectly by identifiers of the foreign entity.

An Aggregate is a collection of entities that are strongly connected to each other. Each aggregate has a root and a boundary. The aggregate root should be the only entity of an aggregate to which entities from outside the aggregate refer. Often, the life cycle of all entities of an aggregate is the same. When the aggregate root is deleted, for example, then all other objects of the aggregate are also deleted. All entities that are directly navigable from an aggregate root are, by definition, within the aggregate's boundary. Associations to entities of a different aggregate cross the boundary between the two aggregates. A boundary means that there is no direct object relationship between the entities but only a reference in form of the identifier of the foreign entity. As a consequence a client typically needs to explicitly retrieve entities beyond the boundary from a repository or a service. Such boundaries exist, for example, because of performance issues or security constraints. Boundaries also exist if data is distributed over multiple data sources.

A Repository is an object that provides facilities to search for entities of a specific type and to load the entities from the data storage or write them back to it. Typically, all database access code of an application is encapsulated by some repository.

A Service is an object that encapsulates business logic or a technical implementation that does not belong to a single entity. Services can be restricted to be accessible only from within the application or they can be exposed to be accessible from remote clients.

Figure 1 gives an example of these concepts in form of a simplified domain model for an online store: a *Purchase Service* exposes functionality to buy products. It relies on two repositories for data retrieval and storage: *Order Repository* and *Product Repository*. Each of these repositories give access to an aggregate: the *Order Aggregate* encapsulates an *Order* with its *Order Items* and the *Product Aggregate* encapsulates a *Product* with its *Product Descriptions*. An order object points to all of its order item objects and a product object points to all of its product description objects. However, order item objects refers to products only by the products' identifiers, i.e., there are no direct object relationships from order item objects to product objects.

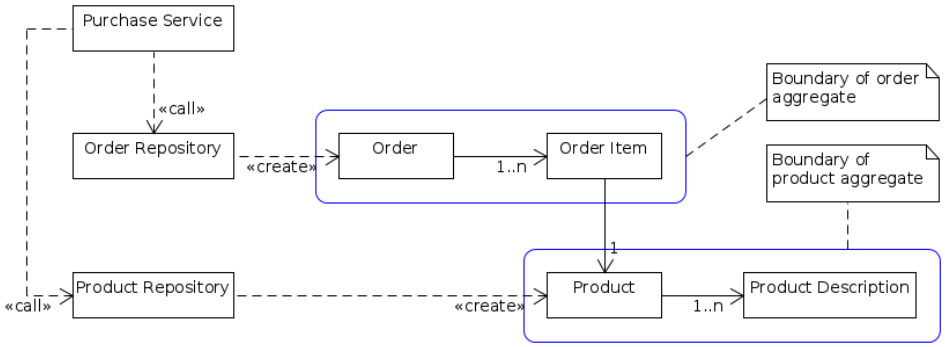

Fig. 1: Example of a domain model

The presented building blocks of domain-driven design make it possible to create complex, yet maintainable software systems. However, they do not focus on non-functional requirements such as performance. Although premature performance optimizations of small efficiencies are considered harmful, critical decisions should be taken upfront [8].

The Entity View pattern adds another building block to domain-driven design that tackles performance issues and some other non-functional requirements such as the complexity of inter-layer communication.

The Entity View Pattern

Suppose that a software application such as an enterprise information system has a complex domain model with many entities and associations between them. Services and repositories provide access to the entities. The entities may be stored in multiple local databases (such as a relational database, a document database and a full-text search engine) or they may be managed by external systems that are accessed by remote service calls.

Also assume that the domain model is part of a layer whose responsibility is to provide data and business logic to other layers within the same application. Clients of the domain layer need access to the application's data for various reasons. For example, the UI layer needs to show information to the application's users. Or the application integrates a full-text search engine that needs to be updated when some data changes.

In general, the domain model allows clients to traverse associations between entities within the entities' boundaries. However, boundaries cannot be crossed by traversing object references. Instead data beyond a boundary needs to be explicitly fetched via a call to a service or repository.

From the perspective of a client, boundaries in the domain model are painful. Every boundary that needs to be crossed by a client results in additional development effort in order to retrieve data beyond the boundary. But technical implications from the data storage or domain model design should not hinder the development of the client.

The following *forces* make it difficult to find a good solution.

- *Domain model design*. Boundaries in the domain model are an essential means to ensure the integrity and stability of the domain model. Experience has shown that small aggregates should be favored against large aggregates [2]. But small aggregates lead to a high number of boundaries.
- *Performance*. The number of round-trips between a client and a service significantly influences the overall performance of the client application. A boundary in the domain model typically results in an additional round-trip to gather the data beyond the boundary. In case of lists of associated entities, the *n+1 selects problem* [6] might apply. In that case one more call is needed for every single associated entity. In addition, if a client issues many small requests for related data then it is difficult to optimize the implementation of the service internally.
- *Complexity*. The more services and repositories are involved in providing data for a client, the more demanding the development of the client becomes: software developers need to know more services and there are more ways to introduce bugs or slow down the application by inappropriate usage of the services. On the service-side, the more services are made available to clients, the more effort is needed to maintain these services in the long run.
- *Security*. Access to some or all of the data may be restricted. Without proper permissions, such data may not be transmitted to the client. Every service that makes restricted data accessible must thoroughly check the client's permissions. On the client side, every service call for restricted data must be carefully designed in order to handle limited access to the required data.

* * *

Therefore:

Introduce an Entity View for every entity whose boundaries hinder client development. Such an entity view is an object that encapsulates both the entity itself (or parts of it) and additional information about the entity that lies beyond the entity's boundaries. Plus, it may contain business logic that operates on the encapsulated data.

An entity view is an object that can be designed as needed to store arbitrary data about an entity. The intention is to keep all information about an entity in a single place to conveniently carry that information anywhere in the application. In general, a client that receives an entity view should get all information that there is about the entity.

Entity views may contain references to any entities of the domain model. Existing data structures can be reused as long as there is no need to hide some data, for example, because of security constraints. This means, an entity view may just be a container with a reference to the entity itself plus references to a couple of related entities. But an entity view may also copy data from entities and incorporate a dedicated data structure if this simplifies data access or increases data integrity.

The aggregation of an entity view's content should be performed by a Service Facade [4], in particular if multiple complex operations are necessary to fetch the required data. According to Deepak Alur, a service facade is meant to “encapsulate and expose business behavior in a coarse-grained manner to the application clients and hide the complexities of business components and their interactions”. A client should only interact with the service facade and retrieve all required data in a single entity view at once. A call to the service facade may thus result in multiple operations, for example to retrieve data from several data sources or even from external systems.

An entity view is typically meant to be read-only. The data is fetched from multiple places and put together into a structure different from its original form. While it is possible to write back changes to entity views, a read-only approach is significantly easier to accomplish. In that respect, entity views are similar to views in a relational database. A database view allows the retrieval of data that is stored in multiple tables via a single operation. The client of such a view does not need to know where the original data is located and how the tables are related to each other. In addition, a view can

provide processed data, i.e., information that does not exist in the same form in any table. However, it is rather difficult to support writing back changes via a view.

The service facade is responsible to check the client's permissions. Depending on these permissions, some pieces of data may be omitted. In that case, the entity view should be designed in a way that all of its data is optional. If a permission is missing, the view just does not hold the respective data.

Introducing entity views into the domain model of an application brings a couple of *advantages*.

- *Domain model design*. The design of the domain model can be focused on the needs of the model itself. Entity views compose an additional layer in the domain model and, to some degree, decouple the demands of clients from the demands of the domain layer.
- *Performance*. Entity views reduce the number of round-trips between clients and the domain layer (and possibly external data sources). In addition, the implementation of a service facade facilitates improving the performance of fetching the required data. As entity views are typically read-only, less transaction overhead may occur when fetching the data from their data sources.
- *Complexity*. An entity view reduces the number of services that need to be accessible to clientss. Client developers need to know fewer services and service developers need to maintain fewer services that are available to other system layers.
- *Security*. All security checks are bundled within the service facade. While implementing these checks might not become easier, the implementation is at least kept at a single place.

However, the entity view pattern also has *liabilities*.

- *Effort*. Entity views are meant to reduce the development effort of the domain layer's clients. Nevertheless, they increase the effort on the domain layer. Depending on the complexity of either side, the additional effort on the domain layer might outweigh the reduced effort on the clients.
- *Performance*. If different clients have different needs in what data they need, the service facade needs to retrieve all data that is needed by any of the clients. Odds are that for every individual call from a client, more data is fetched than needed. Applying a Specification [1] or an Object Prefetch Filter [7] might help in this case but increases the complexity even more. A specification "allows a client to describe (that is, specify) what it wants without concern for how it will be obtained". A specification allows a client to dynamically describe which entities it needs, whereas an object prefetch filter allows a client to dynamically describe which pieces of data is needs per entity.
- *Complexity*. As entity views may contain business logic, there is a risk that the required business logic becomes duplicated from the main entities of the domain model.

The Entity View pattern shares several properties with the Data Transfer Object pattern [5]. According to Martin Fowler, a data transfer object is "an object that carries data between processes in order to reduce the number of method calls" in the context of a remote interface. Data transfer objects are defined together with interfaces that expose parts of the domain model to external systems. They are made to optimize inter-process calls by providing as much data as needed for every such call. But because data transfer objects are expected to be serializable and transported to other systems, they do not contain any business logic and may not contain any data that must be kept confidential inside the application. Data transfer objects are not part of the domain model of an application. Rather, they decouple the external interface from the domain model.

The Entity View pattern is similar to the Data Transfer Object pattern in several ways: data is aggregated and carried according to the needs of some clients. But entity views are meant to be an integral part of the domain model. The can contain business logic, they do not need to be serializable and they may keep references to entities – there is no need to duplicate data structures. Entity views can be used to move aggregated data between layers of an application and processed wherever the data is needed. They do not belong exclusively to the definition of a service.

Entity views only belong to the application itself. They are not part of a published interface that is exposed to external systems. Therefore, if some entities of the domain model change, the entity views may change accordingly. Data transfer objects, on the other hand, are expected to remain unchanged in order to keep public interfaces compatible.

There are several variants how you can use Entity Views.

View per Entity. Create an entity view for every entity whose data needs to be available to a client. This variant is useful if the clients' demands on data vary little. Every client receives the same entity

view when data about an entity is requested. Creating one entity view per entity might lead to large entity views that contain everything that is to be known about the entities while only some aspects of that data is needed for every individual use-case.

View per Use-Case. Create an entity view for every complex use-case. This variant is useful if the clients' demands on data vary a lot. For every use-case, a dedicated entity view contains exactly the data that is relevant to the clients in the context of that use-case. Creating views per use-case can lead to a proliferation of entity views (such as multiple views per entity) but every entity view contains no more data that needed. Therefore, the creation of the entity views are as cheap as can be.

Preview / Detail View. Create one entity view with a small subset of data about an entity and another entity view with a complete set of data. This variant is useful if a client sometimes needs lists of entities with minor details and sometimes needs full data of an entity, such as for a master-detail user interface.

Static View / Dynamic View. Create one entity view that contains all static data, i.e., the information that does not change in the context of the client's interaction. Create another entity view that contains all dynamic data, i.e., that information that does change in the same context. This variant is useful if a relevant part of an entity's data changes during an interaction while another part does not change. By creating two separate views, the data that needs to be fetched after every interaction is reduced to a minimum.

In the domain model of an application, all of these variants can coexist.

Example

A car company develops an online store to sell their cars directly to customers. At the heart of the online store, there are two pages on which a customer chooses a car model and picks additional accessories. The second page in particular shows a lot of information about the chosen car model: the technical specification, pictures, the available accessories, the price to pay, special offers to provide financing and so on. In order to serve international customers, all information needs to be shown in the customer's local language.

The software development team has already set up a domain model that keeps most of the data in a local database: the available car models, their technical specifications and available accessories. However, some data is retrieved from other systems on demand, such as the calculated price and pictures of the vehicles (that must exactly match the chosen accessories).

For the development of the user interface, the team decides to decouple the UI layer from all details how to retrieve data from the internal and external services. Rather, they introduce a service facade that handles the calls to both the domain layer and the external systems. See figure 2 for an overview of the architecture of the online store.

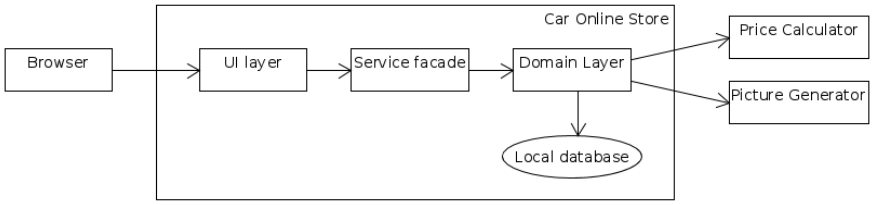

Fig. 2: Overview of the online store's architecture

The domain layer consists of entities that hold data, repositories that retrieve the data from the database, services that implement business logic and technical services that interact with external systems. Figure 3 shows a conceptual overview of the domain model's entities and their boundaries.

A key entity is the *Car Model* with its available *Accessories*. These entities form an aggregate and can be fetched together from the local database. *Car Model Specifications* and their *Car Part Specifications* form another aggregate. *Descriptions* for car models and accessories stand on their own because they exist in many languages but are only needed in one specific language at a time. *Base prices* and *calculated prices* with additional *financing details* are separate aggregates as well because they need to be fetched from an external price calculator. The same applies to *pictures*. They also need to be fetched on demand from an external picture generator. A *car configuration* keeps references to the car model and the accessories that a customer has picked in the online store.

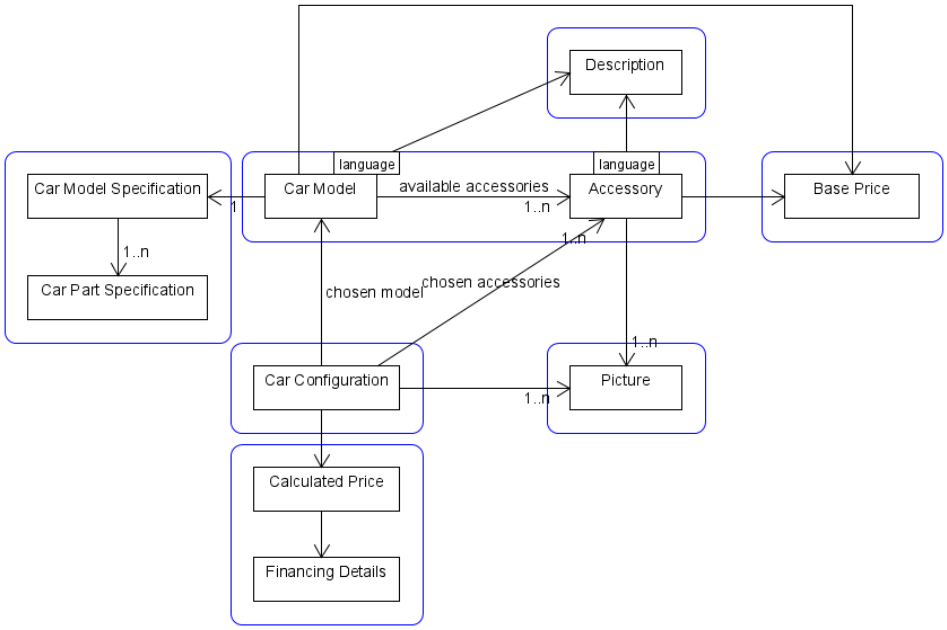

Fig. 3: Entities, their associations and boundaries

Because of the boundaries between the aggregates, the actual domain model implementation does not allow traversing the objects from one aggregate to another. Rather, for every crossing of a boundary, the associated entities need to be fetched explicitly by a call to a repository or a service. For this reason, the domain model also contains such a repository or service for every aggregate root (see figure 4).

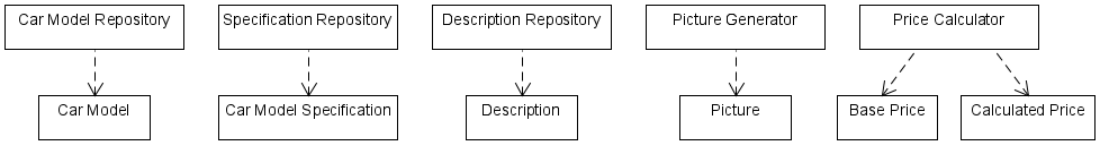

Fig. 4: Repositories and their respective aggregate roots

Now, the development teams needs to decide how the service facade should be designed so that the user interface layer can conveniently access the data. The page that shows the available car models with their default configurations needs only a subset of the available data for a single car model but for several car models at once. Thus, a *Car Model Preview* is added to the domain model (see figure 5). The respective *Car Configuration Facade* retrieves preview objects for all car models, each of which filled only with the most relevant data about the car models.

Note that the preview object contains object references to the domain model entities *Car Model*, *Base Price* and *Picture* but not to a *Description* object. Rather, it contains a copy of the description text in the requested language.

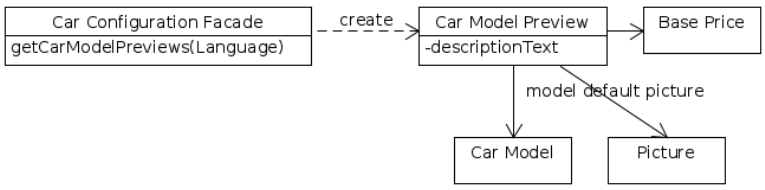

Fig. 5: Car model preview and dependent objects

The page on which a customer configures the car needs very detailed information about the car model. So, a detail view seems like a good way to go. But every action of the customer on the web page may change some of the shown data (e.g., the price to pay, available accessories). Thus, the team decides to create two entity details views: a *Static Car Configuration Detail View* (see figure 6) and a *Dynamic Car Configuration Detail View* (see figure 7).

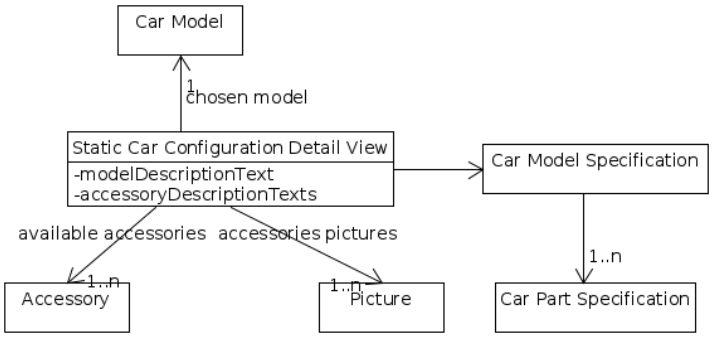

Fig. 6: Static car configuration detail view and dependent objects

The static entity view contains the descriptions for the car model and all accessories in the requested language and holds references to all domain model entities whose data does not change during the configuration of a car.

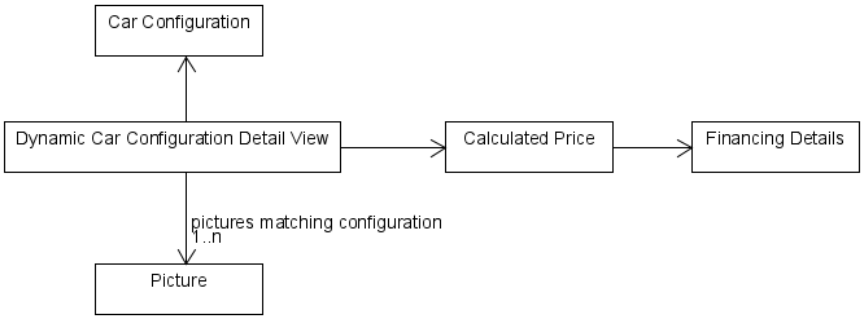

Fig. 7: Dynamic car configuration detail view and dependent objects

The dynamic entity view holds references to all domain model entities whose data changes during the configuration of a car: the price to pay based on the currently chosen accessories and the pictures that match the current car configuration.

In order to render the car configuration page initially, the UI layer loads both the static and the dynamic view once. Every time the customer modifies the car's configuration, the UI layer reloads the dynamic view and updates the user interface accordingly.

Using this approach, the development team quickly succeeds in creating the user interface. However, during the first user tests, a new requirement comes up: a customer should be able to download a car configuration as a PDF document.

A member from another team with great experience in creating PDF documents joins the online store team. After a close look at the service facades and the available entity views, the new member can easily get access to all data that needs to be presented in the PDF document. In-depth knowledge of the domain model and the surrounding external systems is not necessary to start coding.

Conclusion

The Entity View pattern fills a gap in domain-driven design. While the idea behind the pattern (transferring data in a dedicated data structure) is not new, the pattern's focus on non-functional requirements should help application developers to improve the design of their systems.

Acknowledgments

I'm very grateful to my shepherd Christian Köppe who helped me to find a good structure for my paper and gave me many insightful suggestions for improvements. I'd also like to thank the participants of my workshop group at EuroPLoP 2015 for all of their productive comments.

References

[1] Eric Evans: Domain-Driven Design, Addison Wesley, 2004

[2] Vaughn Vernon: Effective Aggregate Design, http://dddcommunity.org/library/vernon_2011

[3] Tim Wellhausen: Business Logic in the Presentation Layer, Proceedings of EuroPLoP 2006, http://tim-wellhausen.de/papers/BusinessLogic.pdf

[4] Deepak Alur, John Crupi and Dan Malks: Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall, 2003

[5] Martin Fowler: Patterns of Enterprise Application Architecture, Addison Wesley, 2003

[6] What is the n+1 selects issue?, https://stackoverflow.com/questions/97197/what-is-the-n1-selects-issue

[7] Tim Wellhausen: Object Prefetch Filter - A Pattern for Improving the Performance of Object Retrieval of Object-Relational Mapping Tools, EuroPLoP 2007, http://tim-wellhausen.de/papers/ObjectPrefetchFilter.pdf

[8] Donald Knuth: Structured Programming with go to Statements, ACM Journal Computing Surveys, Vol 6, No. 4, Dec. 1974. p.268

[9] Martin Fowler: Polyglot Persistence, http://martinfowler.com/bliki/PolyglotPersistence.html