
# DDD: Put logic in service or aggregate root?

Asked 5 years, 11 months ago Active 5 years, 3 months ago Viewed 2k times

2

3

Let's say we are building a document management system.

One project has many documents. I decide to make projects as aggregate root.

If the logic to add one document into project is complicated, I can put it in aggregate root like:

```
class Project
{
    public function addDocument($params){}
}
```

But I can also make this just a domain service

```
class ProjectService
{

    public function addDocument($project, $params){}

}
```

```
}
```

What's the difference?

How to decide which one to use?

Thanks!

domain-driven-design

Share Improve this question Follow

asked Jun 20 '15 at 7:59

尤川豪

**123** 6

1 You can do either, but I think the DDD methodology would favour Project.AddDocument() The downside of ProjectService is you will have to expose and change properties on document and project. I would use the service for cases where no change to the input parameters is required – Ewan Jun 20 '15 at 9:28

2 I agree with Ewan. If `addDocument` changes state of the `Project`, add it here. Otherwise use a domain service (SR principle). – Alexander Langer Jun 20 '15 at 9:38

a nice read Don't Create Aggregate Roots. – Songo Jun 20 '15 at 16:15

@songo Thanks! I will have a read! Thank you very much! – 尤川豪 Jun 23 '15 at 8:26

## 1 Answer

Active | Oldest | Votes

4

Project.addDocument is the right approach.

The guiding principle is the definition of an aggregate.

**AGGREGATE** A cluster of associated objects that are treated as a unit for the purpose of data changes. External references are restricted to one member of the aggregate, designated as the root. A set of consistency rules applies within the aggregate's

boundaries.

In other words, if you have a collection of documents, then all of the consistency rules for the collection of documents belong with the collection, all contained within one aggregate root.

Putting the consistency rules into a domain service tends to lead to an anemic domain model, an anti-pattern.

In `ddd`, all the work of changing state belongs in the aggregate; stateless domain services provide query support to the aggregate that is considering a change.

*Warning: contrived example ahead*

For instance, if you were trying to implement the invariant that a project isn't allowed to include more than $20 worth of documents, then you might see a signature like

```
class Project
{
    public function addDocument($document, $documentPricingService){}
}
```

The pricing service would know how to calculate the price of a document, or even a collection of documents, but it wouldn't know anything about the rule that the total needs to be below $20. The project would know how to pass the (updated) document collection to the domain service for pricing, but not anything about how pricing is done.

Typically, the application component is responsible for finding the pricing service before running the Project.addDocument command.

Share Improve this answer Follow

answered Jan 29 '16 at 4:20

VoiceOfUnreason

**26.7k** 1 34 62