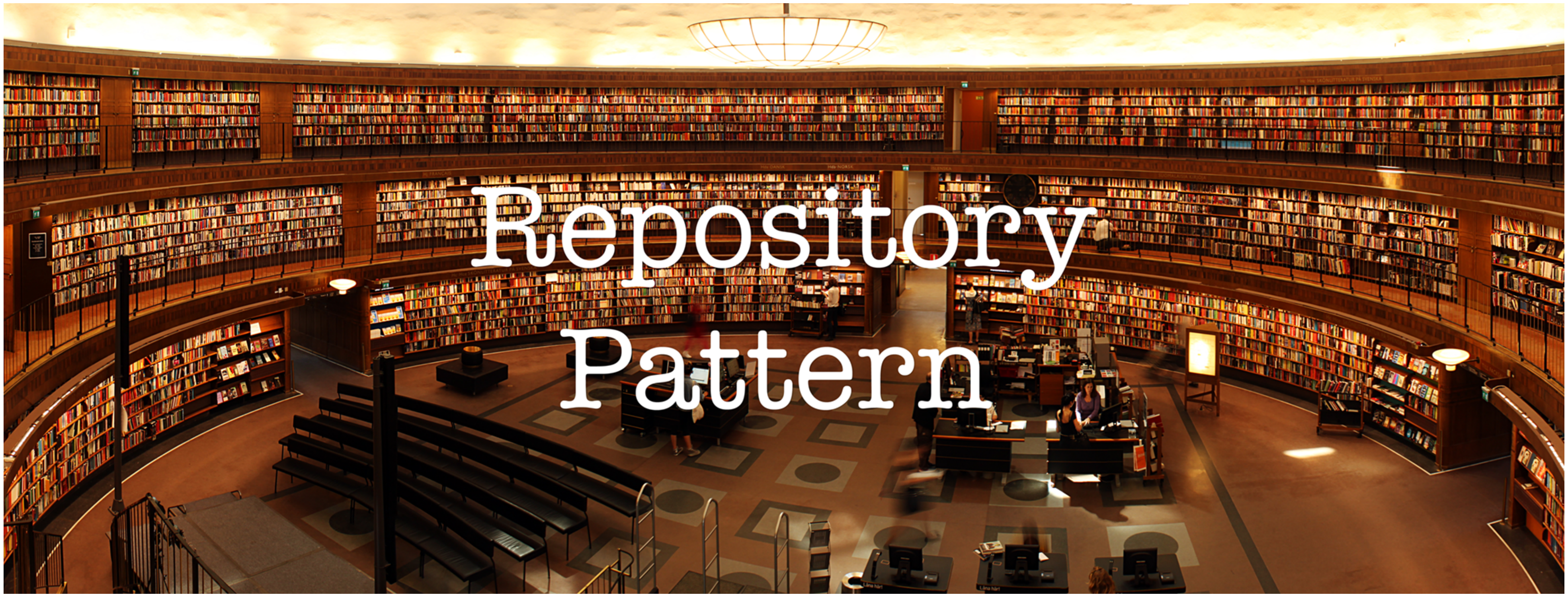
Repository Design Pattern

Per-Erik Bergman Follow
Apr 20, 2017 · 3 min read



The repository pattern is one of the more popular patterns at the moment. I for one like it, it follows the solid principles and done right it is clean and easy to use.

As I see it the repository pattern have two purposes; first it is an abstraction of the data layer and second it is a way of centralising the handling of the domain objects.

Data Access Objects

I still think this is one of the more common ways to work with the data layer. I personal have countless time created DAOs looking like this:

```
public interface ArticleDao {

    List<Article> readAll();

    List<Article> readLatest();

    List<Article> readByTags(Tag... tags);

    Article readById(long id);

    Article create(Article article);

    Article update(Article article);

    Article delete(Article article);

}
```

I don't necessary think this is a bad way, it is structured and fairly easy to use and maintain, but in my stride to write better software I tend to look for better ways of implementing the different parts of my software. DAOs are also closely associated with databases while repositories are not.

My philosophy about coding is about to keep things **small**, **simple** and **consistent**.

I started with one giant DAO handling all my different models. Even if this humungous big class was convenient it was highly clumsy and hard to modify, maintain and almost impossible to reuse.

I started to split it up into several smaller DAO classes just like the one above and they became **smaller** and **simpler** to work with. I also tend to want all DAOs look as similar as possible regarding their methods. You probably can see that the step from my **consistent** way of implementing DAOs to the repository pattern is only one simple interface away (almost).

Abstraction

The idea with this pattern is to have a generic abstract way for the app to work with the data layer without being bother with if the implementation is towards a local database or towards an online API.

The methods are based on the CRUD methods; Create, Read, Update and Delete.

So my first try looked something like this:

```
public interface Repository<T> {

    List<T> readAll();

    T read(Criteria criteria);
    T create(T entity);

    T update(T entity);

    T delete(T entity);

}
```

It is a nice generic abstraction. But let's look at the implementation specially the ***read(Criteria criteria)*** method. Implementing a criteria working with data in memory is simple, let's call it **MemoryCriteria.** Implementing one towards SQLite is fairly simple, let's call that one **SQLiteCriteria**.

The component using this repository have to chose what criteria to use, the criteria is written towards a specific implementation. This breaks the idea of having the app or component working with the repository without knowing about the actual implementation.

Writing one generic criteria that works with all different kind of implementations is not an easy task, so this is not a good way to go. Not for me anyway, I'm way too lazy :-)

Compromise

I chose to compromise and go half way between the repository and my old DAOs. I have my interface with all the CRUD methods. I also add one more method ***readById(K id)***. This interface will cover most of my basic use cases.

```
public interface Repository<T, K> {

    List<T> read();

    T readById(K id);

    T create(T entity);

    T update(T entity);

    T delete(T entity);

}
```

Criteria

We still need a way to get specific selection out of the repository. This is where I compromise.

Say we have a model called Article and we have the need of getting the latest once and the once tagged with certain tags. Let's add another layer of interface containing these methods.

```
public interface ArticleRepository extends
Repository<Article, Long> {
    List<Article> read();

    Article readById(Long id);

    Article create(Article article);
    Article update(Article article);


    Article delete(Article article);


    List<Article> readLatest();


    List<Article> readByTags(Tag...tags);
}
```

Now we have a nice interface to implement to get our repository. It will also be easier to use for the developer since we have clear methods to use instead of a less obvious criteria.