Krzychu Kosobudzki Follow
Mar 20, 2016 · 3 min read

Repository Design Pattern

While developing Android applications in most of them you need to persist data in some kind of storage. In most cases this storage will be SQLite. Probably you want to access data in readable and easy way. What do you do then? Yes… you look for nice looking ORM. You can start to create classes for tables and design all possible method you're likely to use:

```
1  public interface NewsesDao {
2      void add(News news);
3
4      void update(News news);
5
6      void remove(News news);
7
8      News getById();
9
10     List<News> getNewest();
11
12     List<News> getAll();
```

Does it look familiar? Is it wrong? Well, no, but can be better.

Instead of god class (or couple of them) you can follow *Single Responsibility Rule* and use *Repository Design Pattern* instead. Following the definition (http://martinfowler.com/eaaCatalog/repository.html):

> *A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes*

Let's start with base definition of Repository:

```
public interface Repository<T> {
    void add(T item);

    void update(T item);

    void remove(T item);

```

As you can see I replaced Criteria with Specification. Basically it's the same, but Specification seems to be more independent from where you're going to store the data. While using it, I found it extremely useful to add two additional methods:

```
 public interface Repository<T> {
     void add(T item);

     void add(Iterable<T> items);

     void update(T item);

     void remove(T item);

```

It's really what you need. In almost all applications you need to store more than one item and remove a couple of them. RDP should make your live easier, shouldn't it?

There is also *Specification* interface which is only regular marker interface. Once we've defined what the repository is, we can start to use *Specifications*:

```
public interface SqlSpecification extends Specification {
    String toSqlQuery();
}
```

Basic implementation of *Repository* for SQLite can be for instance:

```
public class NewsSqlRepository implements Repository<N
    private final SQLiteOpenHelper openHelper;

    private final Mapper<News, ContentValues> toConten
    private final Mapper<Cursor, News> toNewsMapper;

    public NewsSqlRepository(SQLiteOpenHelper openHelp
        this.openHelper = openHelper;

        this.toContentValuesMapper = new NewsToContent
        this.toNewsMapper = new CursorToNewsMapper();
    }

    @Override
    public void add(News item) {
        add(Collections.singletonList(item));
    }

    @Override
    public void add(Iterable<News> items) {
        final SQLiteDatabase database = openHelper.get
        database.beginTransaction();

        try {
            for (News item : items) {
                final ContentValues contentValues = to

                database.insert(NewsTable.TABLE_NAME,
            }

            database.setTransactionSuccessful();
        } finally {
            database.endTransaction();
            database.close();
        }
    }

    @Override
    public void update(News item) {
        // TODO to be implemented
    }
```

```
42
43      @Override
44      public void remove(News item) {
45          // TODO to be implemented
46      }
47
48      @Override
49      public void remove(Specification specification) {
```

This can be a way simpler by using lambdas or Kotlin ❤. For this example I've deliberately added only implementation of *add* and *query* methods because other methods will look very similar. It's more than sure, you've noticed **Mapper** interface. Its responsibility is to map one object to another. In this case I found it really useful. While using RDP, mapper classes help to keep you focused on what's important.

```
public interface Mapper<From, To> {
    To map(From from);
}
```

To be able to query a database or any other storage we will need an implementation of *Specification*. Here it goes (couple of them, actually):

```
1  public class NewestNewsesSpecification implements SqlS
2
3      @Override
4      public String toSqlQuery() {
5          return String.format(
6                  "SELECT * FROM %1$s ORDER BY `%2$s` DE
7                  NewsTable.TABLE_NAME,
8                  NewsTable.Fields.DATE
```

```
public class NewsByIdSpecification implements SqlSpeci
    private final int id;

    public NewsByIdSpecification(final int id) {
        this.id = id;
    }

    @Override
    public String toSqlQuery() {
        return String.format(
                "SELECT * FROM %1$s WHERE `%2$s` = %3$
                NewsTable.TABLE_NAME,
```

```
public class NewsesByCategorySpecification implements
    private final Category category;

    public NewsesByCategorySpecification(final Categor
        this.category = category;
    }

    @Override
    public String toSqlQuery() {
        return String.format(
                "SELECT * FROM %1$s WHERE `%2$s` = '%3
                NewsTable.TABLE_NAME,
```

Specifications are really simple and in most cases has only one method. You can add as many Specifications as you want, without modifying DAO classes. Moreover testing is simpler. We stick to interfaces, so it can be easily mocked and replaced. You can provide some fake data too, while mapping Repository itself. Can you imagine how those Specification classes are easy to read comparing to veryyyy long DAO?

Repository fits quite good to MVP, but can be easily used in other classes too. By using Dagger you need to define which implementation of Repository you want to use in one place.

```
public class LatestNewsesPresenter implements Presente
    private final LatestNewsesView view;
    private final Repository<News> repository;

    public LatestNewsesPresenter(LatestNewsesView view
        this.view = view;
        this.repository = repository;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        final List<News> newses = repository.query(new

        view.setNewses(newses);
```

Let's complicate it a little bit

OK. Requirements has changed. You can no longer use SQLite. You're cool and therefore you'll be using Realm.

// me waiting for your yelling

What happens when you've implemented application by using DAOs? You have to refactor a tons of classes, even those not related to data access layer. This is not going to happen when *Repository Design Pattern* is your friend, though. All what you need to do is to implement Repository and Specifications to fit new requirements (let's stick to Realm). That's all. You will not be digging in other classes. Take a look at brand new implementation:

```
public class NewsRealmRepository implements Repository
    private final RealmConfiguration realmConfiguratio

    private final Mapper<NewsRealm, News> toNewsMapper

    public NewsRealmRepository(final RealmConfiguratio
        this.realmConfiguration = realmConfiguration;

        this.toNewsMapper = new NewsRealmToNewsMapper(
    }

    @Override
    public void add(final News item) {
        final Realm realm = Realm.getInstance(realmCon

        realm.executeTransaction(new Realm.Transaction
            @Override
            public void execute(Realm realm) {
                final NewsRealm newsRealm = realm.crea
                newsRealm.setCategory(item.getCategory
                newsRealm.setDate(item.getDate());
                newsRealm.setTitle(item.getTitle());
                newsRealm.setText(item.getText());
            }
        });

        realm.close();
    }

    @Override
    public void add(Iterable<News> items) {
        // TODO to be implemented
    }

    @Override
    public void update(News item) {
        // TODO to be implemented
    }

    @Override
    public void remove(News item) {
        // TODO to be implemented
```

```
42            // TODO to be implemented
43        }
```

There is a new Specification, as well:

```
public interface RealmSpecification extends Specification {
    RealmResults<NewsRealm> toRealmResults(Realm realm);
}
```

And an example implementation of just created *RealmSpecification*:

```
1   public class NewsByIdSpecification implements RealmSpe
2       private final int id;
3
4       public NewsByIdSpecification(final int id) {
5           this.id = id;
6       }
7
8       @Override
9       public RealmResults<NewsRealm> toRealmResults(Real
10          return realm.where(NewsRealm.class)
```

Now, you're ready to use Realm instead of SQLite without modifying any unrelated class (do you remember how dependency has been defined in Presenter?).

Bonus

There is also one more extra use case. You can use RDP also to provide caching—as on the previous example, just create *CacheSpecification* e.g.:

```
public interface CacheSpecification<T> extends Specification
{
    boolean accept(T item);
}
```

and use it along with others:

```
1  public class NewsByIdSpecification implements RealmSpe
2      private final int id;
3
4      public NewsByIdSpecification(final int id) {
5          this.id = id;
6      }
7
8      @Override
9      public RealmResults<NewsRealm> toRealmResults(Real
10         return realm.where(NewsRealm.class)
11                 .equalTo(NewsRealm.Fields.ID, id)
12                 .findAll();
13     }
```

What's important, not all of your specifications need to implement all interfaces. If only *NewsByIdSpecification* is used while caching, there is no need to implement *CacheSpecification* by *NewsByCategorySpecification*.

Your code is cleaner now, your teammates love you, kittens not gonna die, and even if you left company the next guy will not kill you, even if he's psychopath.

#ArchitectureMatters