



Project Part 5: Realm-Java

Authors: Wen-Chia Yang, Junxian Chen, Zihua Weng

March 5, 2020

ABSTRACTION

Objectives

In this report, we study five different design patterns and their applications. We identified their corresponding examples in realm-java. Also, follow up the previous homework for the interesting issues, here we contributed our first pull request for one issue in realm-java.

Source Code

Repository: <https://github.com/solution-accepted/realm-java>

Branch: master

RESEARCH

Overview

Design pattern is a general repeatable solution to a commonly occurring problem in software design. They are used to represent some of the best practices adapted by experienced object-oriented software developers.[1] Design patterns can speed up the development process by providing tested, proven development paradigms. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns. [2]

Design patterns are categorized into 3 sub-classifications based on kind of problem they solve. Creational patterns provide the capability to create objects based on a required criteria and in a controlled way. Structural patterns are about organizing different classes and objects to form larger structures and provide new functionality. Finally, behavioral patterns are about identifying common communication patterns between objects and realize these patterns. [1]

Here we will introduce the following design patterns and their according examples in realm-java.

1. Creational:
 - Builder
 - Factory method
 - Singleton
2. Structural:
 - Facade
3. Behavioral
 - Observer

Design Pattern

Builder

Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.[3]

Example:

realm/realm-library/src/main/java/io/realm/RealmConfiguration.java

```
public class RealmConfiguration {
    ...
    private final boolean readOnly;
    ...
    protected RealmConfiguration(...):
    ...
    public static class Builder {
        public Builder() { this(BaseRealm.applicationContext); }
        public RealmConfiguration build() {
            if (readOnly) {
                if (initialDataTransaction != null) {
                    throw new IllegalStateException("This Realm is marked as read-only. Read-only Realms
cannot use initialData(Realm.Transaction).");
                }
            }
        }
        ...

        return new RealmConfiguration(directory,
            fileName,
            getCanonicalPath(new File(directory, fileName)),
            assetFilePath,
            key,
            schemaVersion,
            migration,
            deleteRealmIfMigrationNeeded,
            durability,
            createSchemaMediator(modules, debugSchema),
            rxFactory,
            initialDataTransaction,
            readOnly,
            compactOnLaunch,
            false
        );
    }
}
```

SOLUTION-ACCEPTED

Application :

examples/moduleExample/app/src/main/java/io/realm/examples/appmodules/
ModulesExampleActivity.java

```
RealmConfiguration exoticAnimalsConfig = new RealmConfiguration.Builder()  
    .name("exotic.realm")  
    .modules(new ZooAnimalsModule(), new CreepyAnimalsModule())  
    .build();
```

In realm-java, a RealmConfiguration is used to setup a specific Realm instance. An instance is created by RealmConfiguration.Builder with customized configurations. Since different realmConfigurations are used when initializing the database in different scenarios, using Builder would allow users to create different RealmConfigurations while avoiding telescopic constructor.

Factory Method

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. [4]

Example:

realm/realm-library/src/main/java/io/realm/rx/RealmObservableFactory.java

```
public class RealmObservableFactory implements RxObservableFactory {  
  
    ...  
  
    @Override  
    public <E> Observable<CollectionChange<RealmResults<E>>> changesetsFrom(Realm realm, final  
        RealmResults<E> results) {  
        final RealmConfiguration realmConfig = realm.getConfiguration();  
        return Observable.create(new ObservableOnSubscribe<CollectionChange<RealmResults<E>>>() {...})  
    }  
  
    @Override  
    public <E> Observable<CollectionChange<RealmResults<E>>> changesetsFrom(DynamicRealm realm,  
        final RealmResults<E> results) {  
        final RealmConfiguration realmConfig = realm.getConfiguration();  
        return Observable.create(new ObservableOnSubscribe<CollectionChange<RealmResults<E>>>() {...})  
    }  
}
```

SOLUTION-ACCEPTED

Application:

realm/realm-library/src/main/java/io/realm/RealmObject.java

```
public abstract class RealmObject implements RealmModel, ManagableObject {

    ...

    public static <E extends RealmModel> Observable<ObjectChange<E>> asChangesetObservable(E
object) {
    if (object instanceof RealmObjectProxy) {
        RealmObjectProxy proxy = (RealmObjectProxy) object;
        BaseRealm realm = proxy.realmGet$proxyState().getRealm$realm();
        if (realm instanceof Realm) {
            return realm.configuration.getRxFactory().changesetsFrom((Realm) realm, object);
        } else if (realm instanceof DynamicRealm) {
            DynamicRealm dynamicRealm = (DynamicRealm) realm;
            DynamicRealmObject dynamicObject = (DynamicRealmObject) object;
            return (Observable) realm.configuration.getRxFactory().changesetsFrom(dynamicRealm,
dynamicObject);
        } else {
            throw new UnsupportedOperationException(realm.getClass() + " does not support RxJava." +
" See https://realm.io/docs/java/latest/#rxjava for more details.");
        }
    } else {
        // TODO Is this true? Should we just return Observable.just(object) ?
        throw new IllegalArgumentException("Cannot create Observables from unmanaged RealmObjects");
    }
}
}
```

Realm-java provides a RealmObservableFactory for users to create Observables without manually maintaining a reference to them. This class implements different changesetsFrom methods which return different observable objects according to inputs. In RealmObject class, asChangesetObservable will automatically create the Observable object by calling changesetsFrom methods depends on the realm object. Since RealmObject creates configuration based on the user's input, and the system would not know beforehand, RealmObservableFactory could create the Observable object as needed. When the system improves to allow more configure options, this pattern would help extend the product and maintain the code in an easier way.

Observer

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing. [5] It is used when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.

Example:

realm/realm-library/src/main/java/io/realm/RealmChangeListener.java

```
public interface RealmChangeListener<T> {  
    /**  
     * Called when a transaction is committed.  
     */  
    void onChange(T t);  
}
```

examples/threadExample/src/main/java/io/realm/examples/threads/ThreadFragment.java

```
public class ThreadFragment extends Fragment {  
    // Realm change listener that refreshes the UI when there is changes to Realm.  
    private RealmChangeListener<Realm> realmListener = new RealmChangeListener<Realm>() {  
        @Override  
        public void onChange(Realm realm) { dotsView.invalidate(); }  
    };  
    ...  
    @Override  
    public void onResume() {  
        super.onResume();  
        // Enable UI refresh while the fragment is active.  
        realm.addChangeListener(realmListener);  
        // Create background thread that add a new dot every 0.5 second.  
        backgroundThread = new Thread() {  
            @Override  
            public void run() {...};  
            backgroundThread.start();  
        }  
    }  
  
    @Override  
    public void onPause() {  
        super.onPause();  
        // Disable UI refresh while the fragment is no longer active.  
        realm.removeChangeListener(realmListener);  
        backgroundThread.interrupt();  
    }  
}
```

SOLUTION-ACCEPTED

realm/realm-library/src/main/java/io/realm/internal/ObserverPairList.java

```
public class ObserverPairList<T extends ObserverPairList.ObserverPair> {
    /**
     * @param <T> the type of observer.
     * @param <S> the type of listener.
     */
    public abstract static class ObserverPair<T, S> {
        final WeakReference<T> observerRef;
        protected final S listener;
        // Should only be set by the outer class. To marked it as removed in case it is removed in foreach
        // callback.
        boolean removed = false;
        public ObserverPair(T observer, S listener) {
            this.listener = listener;
            this.observerRef = new WeakReference<T>(observer);
        }
        public void foreach(Callback<T> callback) {
            for (T pair : pairs) {
                if (cleared) {
                    break;
                } else {
                    Object observer = pair.observerRef.get();
                    if (observer == null) {
                        pairs.remove(pair);
                    } else if (!pair.removed) {
                        callback.onCalled(pair, observer);
                    }
                }
            }
        }
        public void add(T pair) {
            if (!pairs.contains(pair)) {
                pairs.add(pair);
                pair.removed = false;
            }
            if (cleared) { cleared = false; }
        }
        public <S, U> void remove(S observer, U listener) {
            for (T pair : pairs) {
                if (observer == pair.observerRef.get() && listener.equals(pair.listener)) {
                    pair.removed = true;
                    pairs.remove(pair);
                    break;
                }
            }
        }
    }
}
```

SOLUTION-ACCEPTED

Realm-java provides `RealmChangeListener` which can be registered with a `Realm`, `RealmResults` or `RealmObject` to receive a notification about updates. For example, in `ThreadFragment`, a `RealmChangeListener` is created and added to `Realm` object. The `Realm` object's observable object and the listener are store as `ObserverPair`. If observable object change, the `foreach` method will be called. But the logic of notification was implemented in `RealmNotifier` class. If any changes happen to `Realm` object, the listener `ThreadFragment` will executed. This pattern help send notifications to existing threads when changes are committed.

Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance. [6] Singleton is often used when the system only allow one single instance for all clients, such as database.

Example:

realm/realm-library/src/main/java/io/realm/Realm.java

```
public class Realm extends BaseRealm {
    ....
    private Realm(RealmCache cache) {
        super(cache, createExpectedSchemaInfo(cache.getConfiguration().getSchemaMediator()));
        schema = new ImmutableRealmSchema(this,
            new ColumnIndices(configuration.getSchemaMediator(), sharedRealm.getSchemaInfo()));
        if (configuration.isReadOnly()) {
            RealmProxyMediator mediator = configuration.getSchemaMediator();
            Set<Class<? extends RealmModel>> classes = mediator.getModelClasses();
            for (Class<? extends RealmModel> clazz : classes) {
                String tableName = Table.getTableNameForClass(mediator.getSimpleClassName(clazz));
                if (!sharedRealm.hasTable(tableName)) {
                    sharedRealm.close();
                    throw new RealmMigrationNeededException(configuration.getPath(),
                        String.format(Locale.US, "Cannot open the read only Realm. '%s' is missing.",
                            Table.getClassNameForTable(tableName)));
                }
            }
        }
    }
}
```

SOLUTION-ACCEPTED

```
public class Realm extends BaseRealm {
    ....
    public static Realm getDefaultInstance() {
        RealmConfiguration configuration = getDefaultConfiguration();
        if (configuration == null) {
            if (BaseRealm.applicationContext == null) {
                throw new IllegalStateException("Call `Realm.init(Context)` before calling this method.");
            } else {
                throw new IllegalStateException("Set default configuration by using  
`Realm.setDefaultConfiguration(RealmConfiguration)`.");
            }
        }
        return RealmCache.createRealmOrGetFromCache(configuration, Realm.class);
    }

    public static Realm getInstance(RealmConfiguration configuration) {
        //noinspection ConstantConditions
        if (configuration == null) {
            throw new IllegalArgumentException(NULL_CONFIG_MSG);
        }
        return RealmCache.createRealmOrGetFromCache(configuration, Realm.class);
    }
}
```

realm/realm-library/src/main/java/io/realm/RealmCache.java

```
static <E extends BaseRealm> E createRealmOrGetFromCache(RealmConfiguration configuration,
    Class<E> realmClass) {
    RealmCache cache = getCache(configuration.getPath(), true);

    return cache.doCreateRealmOrGetFromCache(configuration, realmClass);
}
```

In realm-java, Realm instances are per thread singletons and cached. Realm class provides a `getInstance` method for retrieving the the Realm instance from `RealmCache` based on the configuration. For each configuration, there is only one Realm instance. Therefore, the realm instance could be shared within the system, read and written by all clients.

SOLUTION-ACCEPTED

Facade

A facade is an object that provides a simplified interface to a larger body of code, such as a class library.[7] Facade could also decouple the subsystem from clients and other subsystems.[8] or structure the subsystem into layers.

Example:

realm/realm-library/src/main/java/io/realm/internal/ObjectServerFacade.java

```
public class ObjectServerFacade {

    private final static ObjectServerFacade nonSyncFacade = new ObjectServerFacade();
    private static ObjectServerFacade syncFacade = null;

    static {
        //noinspection TryWithIdenticalCatches
        try {
            @SuppressWarnings("LiteralClassName")
            Class syncFacadeClass = Class.forName("io.realm.internal.SyncObjectServerFacade");
            //noinspection unchecked
            syncFacade = (ObjectServerFacade) syncFacadeClass.getDeclaredConstructor().newInstance();
        } catch (ClassNotFoundException ignored) {
        } catch (InstantiationException e) {
            throw new RealmException("Failed to init SyncObjectServerFacade", e);
        } catch (IllegalAccessException e) {
            throw new RealmException("Failed to init SyncObjectServerFacade", e);
        } catch (NoSuchMethodException e) {
            throw new RealmException("Failed to init SyncObjectServerFacade", e);
        } catch (InvocationTargetException e) {
            throw new RealmException("Failed to init SyncObjectServerFacade", e.getTargetException());
        }
    }

    public Object[] getSyncConfigurationOptions(RealmConfiguration config) {
        return new Object[12];
    }

    ...
}
```

SOLUTION-ACCEPTED

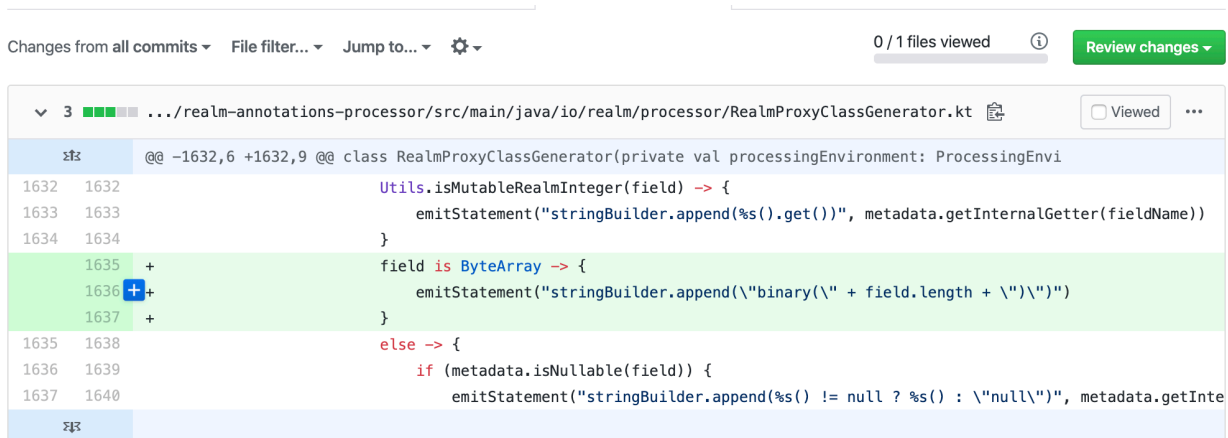
```
public class ObjectServerFacade {  
    ...  
  
    public static ObjectServerFacade getFacade(boolean needSyncFacade) {  
        if (needSyncFacade) {  
            return syncFacade;  
        }  
        return nonSyncFacade;  
    }  
  
    // Returns a SyncObjectServerFacade instance if the class exists. Otherwise returns a non-sync one.  
    public static ObjectServerFacade getSyncFacadelfPossible() {  
        if (syncFacade != null) {  
            return syncFacade;  
        }  
        return nonSyncFacade;  
    }  
  
    // Do nothing  
}
```

Here, ObjectServerFacade is a mediator between the basic Realm APIs and the Object Server APIs. It breaks the cyclic dependency between ObjectServer and Realm code.

PULL REQUEST CONTRIBUTION

ISSUE #1650

As for our first contribution to realm-java, we started from looking issues with First-Good-Issue tag and decided to fix #1650 issue (RealmProxy#toString() should print length information for binary field) [9]. The idea was to add logic to return the length of the Array input for RealmProxy#toString() method. According to the contribution instruction [10], we need to create our own branch for new, signed up the [Contributor License Agreement](#) (CLA) and then create pull request. We finished the logic in RealmProxyClassGenerator.kt with the following code and create a pull request as indicated. (pull request link: <https://github.com/realm/realm-java/pull/6767>)



Changes from all commits ▾ File filter... ▾ Jump to... ▾ ⚙ 0 / 1 files viewed ⓘ Review changes ▾

```
@@ -1632,6 +1632,9 @@ class RealmProxyClassGenerator(private val processingEnvironment: ProcessingEnvi
1632 1632         Utils.isMutableRealmInteger(field) -> {
1633 1633             emitStatement("stringBuilder.append(%s().get())", metadata.getInternalGetter(fieldName))
1634 1634         }
1635 1635         field is ByteArray -> {
1636 1636             emitStatement("stringBuilder.append(\"binary(\" + field.length + \")\")")
1637 1637         }
1638 1638     else -> {
1639 1639         if (metadata.isNullable(field)) {
1640 1640             emitStatement("stringBuilder.append(%s() != null ? %s() : \"null\")", metadata.getInternalGetter(fieldName))
1641 1641         }
1642 1642     }
```

REFERENCE

- [1] https://en.wikipedia.org/wiki/Software_design_pattern
- [2] https://sourcemaking.com/design_patterns
- [3] <https://github.com/iluwatar/java-design-patterns/tree/master/builder>
- [4] <https://refactoring.guru/design-patterns/factory-method>
- [5] <https://refactoring.guru/design-patterns/observer>
- [6] <https://refactoring.guru/design-patterns/singleton>
- [7] https://en.wikipedia.org/wiki/Facade_pattern
- [8] <https://github.com/iluwatar/java-design-patterns/tree/master/facade>
- [9] <https://github.com/realm/realm-java/issues/1650>
- [10] <https://github.com/realm/realm-java/blob/master/CONTRIBUTING.md>