Creating proxies for data objects

One way to have more control over updates is to create proxies for all objects retrieved through the ORM layer which intercept all calls to the entity object. By intercepting these we can know exactly what data is accessed and what data is changed, and this could help with easier models.

Making the proxies themselves is not hard. DomUI already uses the QDataContext/QCriteria framework for all database access, so it is very easy to layer a "special" QDataContext above the real one. The special one acts as a delegate to the original but creates suitable proxies for all entity instances retrieved. These proxies are then returned instead of the originals.

Making the proxies themselves requires the help of a bytecode library like cglib (old), javassist or bytebuddy. There is a Proxy class in the JRE which can make dynamic proxies but as usual it's severely handicapped: it can only create proxies for interfaces, not for concrete classes - so it is useless for this task.

Why proxies?

We want proxies so that we can do one or more of the following:

- Add extra information to entities so that we can keep track of where they originate from (for instance: this QDataContext, this Conversation / screen). This helps with giving proper error messages if objects obtained through one QDataContext are saved/updated through another context.
- We can alter the actual values returned by proxy properties, for instance ensuring that all Lists on the thing are Observable.
- We can automatically handle parent/child association:
    - If a child ManyToOne property is set to a given parent value we can ensure that the parent's OneToMany List will also contain the child.
    - If a child is added to a parent's OneToMany property we can automagically set the corresponding child's ManyToOne parent property.
- We could use it to track changes: we can collect changes in the proxy, and use that to determine what to update - or to rollback.

Most of these are not too hard. The last one, however, needs a bit of further thought.

Using proxies to detect changes for "subtransactions"

We could use a proxy to implement "subtransactions", the idea that the original objects are only changed when the subtransaction "commits". These original objects can then be saved by some other (real) transaction - or even another subtransaction. This allows for layered screens where for instance a Dialog can be used to edit some thingy without those changes becoming permanent even when the dialog is cancelled.

There are two ways to do this:

- The proxy itself keeps copies of all original values and returns those copies when getters and setters are called. The wrapped (original) object remains onchanged. Only when the context is committed will the data be copied to the original. This is the "pessimistic" approach: we assume that shit will happen and will not change anything until we're pretty sure.
- We can let the proxy make copies of all of the original data and then propagate all changes to the original immediately. Only when we rollback do we reset all originals by restoring the original data stored. This is the optimistic approach: what could *possibly* go wrong?

The latter one is undesirable for the following reasons:

- If something goes wrong you need to do the right thing in all circumstances (rolling back) or the compromised/wrong data gets propagated to the real model anyway. As developers are idiots this is a Bad Plan(c).
- Changing the original also means that other parts of the screen (which show data from the original) will change. This is confusing and performance-wise expensive.

So we get rule 1: **data is kept in the proxy and only copied back to the original at special times**

Handling proxy data

Key concept: intercept all getters and setters, and where needed replace the data gotten with different copies. We can use this to replace lists with ObservableLists and ManyToOne relation properties to a proxy of that parent.

Replacing OneToMany Lists with ObservableLists

Take for example the wish to wrap all List properties into an Observable list, so that any change made to it propagates to the UI immediately. This can be done relatively easily:

- As soon as the getter on the proxy is called:
    - Check if we already have a copy of the Observable list for the value, if so return that
    - Call the getter in the original. This will return a (usually lazily loaded) List.
    - Wrap the returned list into a special Observable list which obeys the laziness of the original list: only access the original lists's methods if methods on the ObservableList are called.
    - As soon as the original list is instantiated by a call: get all members and wrap them inside a proxy, so that they themselves are now properly proxied.

We can do something similar when the property's setter is called: if it is called with a non-observable list we can wrap it and put that into the property. This will have side effects though: if the original list is changed after the observable is created this will not send events (as the observable list does not see the changes since they are not made through it). This however should not be an issue in most cases: setting a whole new list will cause all bindings to refresh completely, and this is done after the logic has made the changes; hence no change events are needed for this case.

Replacing ManyToOne (parent) property values

We want to make sure that all entities reachable from the proxy are themselves proxies. We register these with the delegate context so that each entity instance is associated with only one proxy. This creates an "access map" of data: all data accessed through the context will become known, and this limits the data we need to check for changes at commit time.

Problems caused by using proxies

As long as the Entity class contains only getters and setters directly mirrored by fields this approach should work fine. But it goes south very quickly when the entity object contains other methods. Take for instance an entity class that has a toString() method which renders the values of its fields as a string. There are two ways to handle this:

- We can delegate to the original. But the values in the fields for the original might differ from the ones in the proxy (for instance the proxy might have the ObservableList with wrapped entities - and that list might be different).
- We can use the proxy's version. As long as that version calls getters only it would work. But if it accesses fields it does not: these fields in the proxy itself are unchangeable because we've intercepted the setters and getters, so we cannot usually set them to any values 8-/ Hence the proxy's toString() uses empty fields resulting in nothing nice.

The only way around this seems to be the following:

- We need to limit what can be done with getters and setters. Specifically we need getters and setters to be backed by a field and **the same field**.
- Instead of just "overwriting" the setter and getter for properties we need to rename the original ones so that we still have the original accessors to whatever fields are used **inside the proxy**.

The latter solves a lot of issues. We can now store the proxy data inside the real fields of the proxy, and this means that any method inside the proxy that uses those fields will now work **and** use the correct (replaced) data. This also saves us from the need to create separate storage for the fields in the

What needs to be generated as a proxy

For a class Orig we need to generate a class OrigNNNN extends Orig. We should abort if Orig is final - or if it contains final members (other than the finals from Object).

The OrigNNNN class will have extra fields generated to keep data:

Field	Type	Description
__handler	IQProxyDataHandler	The data handler for the proxy which receives all delegated calls, and which contains a reference to the QDataContext and other related data.
__original	Orig	The pointer to the original, wrapped instance.

In addition the proxy contains all fields from Orig because it extends Orig. So the start of the class looks like this:

```
public class Orig$sdads1212 extends Orig {
  private IQProxyHandler __handler;
  private Orig __original;
```

Getters and setters

For each getter T getXxxx() we generate a new accessor method as follows:

```
public T __getXxxx() {
  return super.getXxxx();
}
static private Method __getXxxxM = (initialized to the above method)
```

We do the same for all setters: void setXxxx(T value) generates:

```
public void __setXxxx(T value) {
  super.setXxxx(value);
}
static private final Method __setXxxxM = (initialized to the above method)
```

These are generated in the proxy, and they are needed so that we can actually change the backing fields of the *proxy* when we need to.

For each of these getters and setters we will also generate static private fields containing Method references to these getters and setters: the __setXxxxM and __getXxxxM fields.

The new getter method delegates immediately to the handler:

```
static private final __getXxxxO = (initialized to Orig.getXxxx);
public T getXxxx() {
  return __handler.onGet(this, __original, __getXxxxO, __getXxxM);
}
```

The values passed to the handler allow access to both the proxy and the original, and using the two method references you can get data from both the original (using __getXxxxO method reference) or from the proxy (using __getXxxxM, which uses the "redirected" getter accessing super.getXxxx()).

Something sneakily similar is generated for the getter:

```
static private final __setXxxxO = (initialized to Orig.setXxxx)
public void setXxxx(T value) {
  __handler.onSet(this, __original, __setXxxxO, __setXxxxM, value);
}
```