Investigating updates

One of the hardest parts of writing an UI is to keep track of updates and changes to whatever we're showing inside the UI. Ideally the UI is just a "wrapper" around some stateful data model, and changes to the model are automatically propagated in the correct way to the UI (the view). This is what is (mostly) done with DomUI data binding - but there are lots of things that still need manual labor. I would like to find ways to minimize that labor.

This document is meant as something to order my thoughts.

Changes inside a model

Simple stuff: property data binding

DomUI data binding works by connecting some property on a model to the value of a component. The component or model is updated depending on whether the value changed, which is usually done by Objects.equals() between both values - which in turn calls equals on either object. Once the comparison returns false the whole component is invalidated and redrawn.

Binding like this works OK except for the following cases:

- Large structures like Lists. Calling equals on lists need to compare all elements, and in the most common case (nothing changes) this takes O(n) time every server entry and exit. Solution: make the binding to an ObservableList, and use list evens to handle updates inside the list. This requires special knowledge about the value in the component, but most components are quite aware that they are made for showing lists - so that is doable.
- Binding to *elements* of a larger structure (like a list). Naive code could do something like:

```
for(Person p: personList) {
  add(new DisplayValue().bind().to(p, "fullName"));
}
```

  This creates a bound control for every instance *inside* the list. So if the fullName property of a specific instance changes then the change will be visible on-screen. But if the *list* changes nothing happens- the actual binding is between instances that once were in the list. This, too, requires knowledge of the fact that a list is used.

All in all data binding works well if the objects obey either of the following:

- They properly implement equals (on their members too) and are not large (equals finishes rapidly).
- They are *immutable*. This works well even for large objects because we can use *reference equality* (comparing that the thingies are the same instance) to compare them. FIXME DomUI should implement a generic method to handle these comparisons that take immutability into account, for instance by checking whether the class has some annotation that ends in Immutable.

DomUI properly implements most of this without problems.

Binding to lists

Binding to lists is harder, as stated before. Components that accept a list (either as a value property but also for other bindable properties, like the "data" property on ComboXxxxx combo boxes) have a few choices to handle them:

- Fully compare the lists using equals. While technically correct this is never done because it is way too expensive.
- Assume that the list is immutable and that a new *instance* will be set when the list value changes. This means that reference equality is used in compares. This of course fails if the list contents is changed anyway - there is no way the component can see that. We can add some simple safeguards like saving the size of the list every setValue() and using that in the comparison too. But this crude solution only works for adds and deletes; a modification of a list value is not seen.
- Use an IObservableList based list. This is implemented in for instance DataTable, where the setList() checks whether the list passed is IObservable. In that case the control registers change listeners on that instance, and so it will receive change events from that List. This works quite well but is quite a bit of work to implement inside components. But one way that usually works well is to just invalidate (forceRebuild) the component for every list change. This will cause a re-render but assuming that list changes are not that common this is usually not an issue.

Instrumenting?

Not having proper list events is a real problem, sadly enough, and I see no other feasible solutions. It would be great if we could somehow instrument the list implementations so that they would start collecting change information- but that would mean having to add instrumentation in a separate module which should then change the byte code for the common collection classes we want to be able to check.

This, however, is too much like black magic to interest me.

Updates of database (ORM) data

When we edit data from an ORM we have a specific set of trouble:

- When one part of the screen updates data - how do other parts of the screen know? And how do they update?
- If we edit and cancel - how do we make sure no changes are seen?

These are the basic issues, but many more can be found that are related. What solutions can we find?

The elephant in the house: write models, damnit

There is actually a proper solution to most of these issues which is this: **do not use the ORM data objects directly but always create a model which is a editable copy of that model**. All operations and all business logic **must** then be defined in terms of this model; the only time the actual ORM objects are used are when retrieving data and when saving data.

This works all the time because we fully control and know what is being done, and it is easy (through careful coding) to know what data changed or was needed.

We however dislike this because making models is a lot of work and involves a lot of copying of concepts - especially for the most common (CRUD) screens. For these we want a solution that works quickly without extra typing or copying of code.

Any complex data manipulation should however always be done using handwritten models.

Let's start with issues and possible solutions.

Common ORM related issues

TODO Discuss:

- Passing updates between screen instances for the same user (screen stack / subscreens)
- Handling changes when dynamic queries determine the data shown (and a change would cause that query to show new/changed data).
- Handling child relation List updates. When something is added somewhere that should be present in a parent's list-of-children - how can we do that?

Edit dialogs and cancel/rollback

Another class of update trouble has to do with edit dialogs and what happens when we cancel them 8-/. When we use data binding all changes made in the dialog will be propagated to the model. This is all fine and dandy when we finally decide to save those changes: this would persist them and the changed values is what we want to have.

But if we change data and decide to cancel we're in trouble: we do not want those changes to be persisted and all changes must be undone.

There are several basic solutions to handle this.

Separate edit screen with separate QDataContext (session)

One way (often used in DomUI 1.0 code) is to separate the edit data by opening a new screen (new UrlPage, new URL). This new page uses a new QDataContext and loads a copy of the data inside the screen. If the screen is cancelled then the QDataContext is rolled back (ensuring that no updates enter the database) and the code goes back to the previous page. This previous page reloads (by devious means like the onShelve event) itself from the (unchanged) database and so no data changes where on rollback.

This works like a charm as long as we have a solution for updating the other pages when save *is* pressed, and when it is acceptable to have full page updates.

FIXME More details on this method will be written in a separate document later which I will link to here once done.

Using SubPage and notifications

A SubPage (DomUI 2.0) is a page like structure that can be used on an UrlPage. An UrlPage can contain multiple SubPage objects and indeed SubPage objects themselves can be nested too.

A SubPage has its own QDataContext which is inherited by all nodes added to the SubPage, and so objects loaded by a subpage are copies inside that SubPage.

With this it is possible to create a single page interface which acts mostly as the 1.0 "Separate edit screen" solution:

- Make a base class which extends SubPage, and let this base class fire Notifications as soon as data is saved.
- Have the page also implement a notification Listener which will call whatever ORM calls are needed to update data that other SubPages have saved.

As long as the data model is not too complex (CRUD++) this works technically very well: cancelling data is no more work than just destroying the SubPage; the underlying QDataContext will then rollback and all changes go to that big bitbucket in the sky.

In practice however this is very hard to work with: a single object tree (the page tree) contains objects loaded in a lot of different QDataContext instances and it is depressingly easy to make a mistake in matching object with the QDataContext it belongs to. A simple case is for instance adding a SubPage which has a Listener which is listened to by the parent of that SubPage. If an object is passed in that listener it will most probably be connected to the child subpage's QDataContext. Any attempt to do something (like saving) in the parent page will cause no end of exception fireworks. This is already bad, but tracking what went wrong is often hours of work because of the many QDataContext instances and the difficulty of knowing which object attaches to which QDataContext - if at all; problems with detached objects come from a special place in Hell reserved for the worst of people (like politicians).

Possible alternative solutions

Copying models

Instead of letting the database handle the problem we could try to do it ourselves - by making a copy of the data we want to edit. We would edit the copy inside the dialog or subscreen and only when we press some save option would we copy back the copy inside the real model- and save that (we would however still be in deep sheet if the save failed - this would destroy Hibernate's session and hence next saves become unreliable).

Copying is technically not too complex, but it quickly breaks when we consider some common cases.

Problems with object trees

If we are not just editing a single instance but also data branching from it (either parent relation objects or child list objects) then it becomes a challenge: which objects do we copy and which not? It is not easy to find this out automatically, and so the obvious solution would be to specify, at copy time, which branches from the root object(s) need to be copied. The idea is more or less to specify all properties and property paths which need to be "deep copied"; all data outside of that path would have shallow copies.

While this works it is a maintenance and debugging nightmare: any change to some code inside the screen which poops outside the copy will change data that it is not meant to change, and this leads to very hard to find bugs.

Problems with Business Logic

The models edited in screens use business logic (like controllers) to do complex changes. For instance a booking screen might move an amount by calling business logic to effect the actual move. Business logic should be a black box (well, considering the code found in them they might be brown boxes, but well) and we should not (need to) know what data objects are changed by that logic. But as we need to somehow specify which objects in the data tree need to be copied this does not work- we can only know what to copy if we know exactly how each logic part operates. And that again makes the code very fragile.

Copying through a "Subsession model"

The fundamental theorem of Software Engineering states: *All problems in computer science can be solved by another level of indirection*. So let's throw *that* on this problem.

A non exhaustive list of things we would like to have is:

- To properly control what is saved and what is updated we need to know what is being *used* in the screen's code and its attached logic - automatically.
- Any object that gets used will be copied before, and the copy is the actual thing changed.
- Alternatively, using things we should automatically get "before copies" so that we can more easily deduct what has really changed.
- It would also be great if we could make sure that all Child lists in objects are Observable, so that any UI logic binding to it updates on changes.
- If we decide to save we would propagate all changes from the copies to a (possibly fresh) set of data objects and commit. This would also save us from Hibernate session trouble: if the save fails then we just discard the session; the next save would reload everything anew and hopefully fare better.

One way that we can probably do this is by using the existing QDataContext abstraction. This is already used for all data access so it is a central point where all data obtained from the database comes from. It does not even need to be ORM data: we have QDataContext implementations that can use JDBC too, and it is not too hard to write other implementations.

Wrapping QDataContext

We could create a "SubSession" QDataContext which "wraps" another QDataContext. This is the first of the "indirection" layers we need. The subsession context would mostly act as a delegate for the wrapped QDataContext, but it would have some tricks up its sleeve.

All queries (or other methods that would return new data classes) will register the originals inside a "source map". Each object will then be copied and wrapped into a dynamic proxy. The dynamic proxy "extends" the original data class but intercepts all getters and setters (at the very least those for relation properties). As all objects are proxied it is easy to expose specific helper interfaces on those proxies so it becomes easier to control (and show) what data came from where, and where it belongs to. This means we can give way more meaningful error messages if objects are mixed up in data contexts.

The proxies intercept getters so that they can properly register the result of the get, i.e. if a getter returns a lazily loaded instance we can wrap that instance too and register it, so that we can save it when needed.

Stream of conscience - endless, endless delta's

An alternative to keeping before images is to actually change the processing model itself. Instead of collecting values into the objects we use the initial state of these objects as read-only data; from there on all changes made are not made to the objects but to a supermodel on top of them, and this supermodel collects the changes as a train of delta's. While a screen is edited the delta's are collected, and to show values the code rolls forward all delta's for a given field starting from the base value.

This has the advantage of being easily able to rollback to any state in the past, so it can be used to things like undo and rollback easily.