Issue Custom Fields Parent/Child:

I am a newbie trying to do this requirement. I could not make it work by just using the plug-in. It seems like many of details solution is within the Redmine core. HELP!

Our project requires the use of *Issue Custom Fields* model a parent/child type relationship. Currently, there is no facility to define both `Parent` and `Child` as entity classes with a < *bidirectional* `one-to-many`> or mostly < *bidirectional* `one-to-one`> association *with cascades* to model a parent/child relationship. A very simple example is;

Building [Main, Engineering, Production, …]

Has Floor [1, 2, 3, 4, 5, 6, …]

Has unit [1, 2, 3, 4, 5, …]

Has room [bedroom, washroom, dining, …]

Has flooring [hardwood, carpet, cement, …]

The chain of parent-child will look like this:

Building (Main) ➔ Floor (1) ➔ Unit (101) ➔ Room (living room) Flooring (hardwood)

Here is the list of my pain points:

1. Behavior of the field list collections

List collections are considered to be a logical part of their owning entity and not of the contained entities. Be aware that this is a critical distinction that has the following consequences:

- When you remove/add an object from/to a list collection, the version number of the collection owner is incremented.
- If an object that was removed from a list collection is an instance of a value type (e.g. a composite element), that object will cease to be persistent and its state will be completely removed from the database. Likewise, adding a value type instance to the list collection will cause its state to be immediately persistent.
- Conversely, if an entity is removed from a list collection (a one-to-many or many-to-many association), it will not be deleted by default. This behavior is completely consistent; a change to the internal state of another entity should not cause the associated entity to vanish. Likewise, adding an entity to a list collection does not cause that entity to become persistent, by default.

Adding an entity to a list collection, by default, merely creates a link between the two entities. Removing the entity will remove the link. This is appropriate for all sorts of cases. However, it is not appropriate in the case of a parent/child relationship. In this case, the life of the child is bound to the life cycle of the parent.

For example, `<one-to-many>` or `<one-to-one>` can be done by doing an `INSERT` to create the record for `c and` an `UPDATE` to create the link from `p` to `c` and on the `parent_id` column, specify `not-null="true"` in the collection mapping. And also add the `parent` property to the `Child` class. Now that the `Child` entity is managing the state of the link, we tell the collection not to update the link.

2. Cascading life cycle

Similarly, we do not need to iterate over the children when saving or deleting a `Parent`. The following removes `p` and all its children from the database.

However, deleting c will not remove `c` from the database. In this case, it will only remove the link to `p` and cause a `NOT NULL` constraint violation. You need to explicitly `delete()` the `Child`.

The right way, a `Child` cannot exist without its parent. So if we remove a `Child` from the collection, we do want it to be deleted.

Even though the collection mapping specifies `inverse="true"`, cascades are still processed by iterating the collection elements. If you need an object be saved, deleted or updated by cascade, you must add it to the collection.

3. Cascades and unsaved-value

Suppose we added up a `Parent` in one `Session`, made some changes in a UI action and wanted to persist these changes. The `Parent` will contain a collection of children and, since the cascading update is enabled, it is necessary to know which children are newly instantiated and which represent existing rows in the database. We will also assume that both *`Parent`* and *`Child`* have generated identifier properties of type *`LIST`*.

This may be suitable for the case of a generated identifier, but what about assigned identifiers and composite identifiers? This is more difficult, since Redmine cannot use the identifier property to distinguish between a newly instantiated object, with an identifier assigned by the user, and an object loaded in a previous session. In this case, Redmine will either use the timestamp or version property, or will actually query the second-level cache or, worst case, the database, to see if the row exists.

4. Conclusion

None of the above issues exist in the case of `<composite-element>` mappings, which have exactly the semantics of a parent/child relationship. Unfortunately, there are two big limitations with composite element classes: composite elements cannot own collections and they should not be the child of any entity other than the unique parent.

Just like all issue custom fields, they can be tracked, required, used for filter, and searchable.