OPEN SOURCE APPLICATIONS FOUNDATION

Status Update Number 7 -- June 25 2003

1. Design work.

We're still deep in the design phase, making incremental progress on the key infrastructural elements. Brian is working on creating a formal representation of the data model. Once it exists we'll circulate it. Right now we have notes of various discussions at http://wiki.osafoundation.org/bin/view/Main/DataModelFeatureDetails, but no cohesive design document. Discussions of the "document architecture (formerly the "super-widget" now sometimes know as the "Table and Outline Widget") continue. We'll be focusing on getting the useful parts of these discussions written down soon, as soon as we figure out how to clone Brian's amazing ability to document meetings, discussions and progress. Andy Hertzfeld and Andrew Francis have been working on the API for the notification framework for the event model. It looks like the API is starting to coalesce, and implementation of the Notification Manager http://wiki.osafoundation.org/bin/view/Main/NotificationModel may start soon.

2. Competitive Anaylsis.

We now have a competitive survey and analysis of email and calendar applications in the Higher Education market. This work has been done primarily by Ed Chao, who's volunteered his help in the project management arena over the summer. Many thanks to Ed.

3. 0.2 scheduling.

We're still working on it, both on a bottoms-up approach of planning specific tasks, and using a top-down approach of estimating general functionality. No breakthrough news.

4. Project pages.

As part of our overall planning and organizational goals, we've decided to create "project pages" for a handful of high-level pieces of chandler (such as calendar, data model and agents). Our first goal is to create an Overview Page, listing the approximately 30 or so areas which together make up our view of Chandler, and to actually create Project Pages for the 5 or 6 most pressing areas. The project pages will ultimately provide an overview of the relevant piece of Chandler. Mtoy will create the Overview page.

5. Email

Ducky is putting together information about existing email libraries so we can figure out whether we'll use existing libraries. We also need to look at the licenses that govern them and make some decisions about our licensing plans. See the Implications of Third Party Licenses doc for a discussion of this. Ducky is also working on estimating the time necessary for getting email working in chandler if we build it all ourselves and if we use existing libraries. Her thinking so far can be found at:
http://wiki.osafoundation.org/bin/view/Main/EmailSchedules. If you have releavant experience and something to add about scheduling, then please do so.

6. Implementation

Andi (with an "i") is working away madly. He's working on finalizing the XML format and an implementation of the schema of schemas, and having code in CVS which developers can use to build schemas on. You can now create objects, save them and get them back. Andi has also created a Chandler Data Model Developer Guide can be found at http://wiki.osafoundation.org/bin/view/Main/DataModelDevGuide. The code is new and still unpolished, so expect some surprises:-)
Jed's been working on the ActionsBar? and item detail view for new windows.

On the build front, Morgen has startes a smoketests page at http://wiki.osafoundation.org/bin/view/Main/SmokeTests and a unit test page at http://wiki.osafoundation.org/bin/view/Main/UnitTests. Morgen has merged the newest wxPython release into the Chandler CVS repository. Morgen also debugged an OS X problem that caused our Mac builds to be broken for a few days. Builds should be OK now.

7. Hiring.

We still have open positions which we need to fill. Many thanks to David Paigen for his assistance with job descriptions, postings and resumes. We've now identified the hiring issue as a "snake": posted positions remain unfilled for long time, we don't have an agreed upon, effective hiring process, and it's a big priority. We decided to review at our hiring process to see if we can do better.

-- MitchellBaker - 26 Jun 2003

Referenced Content

Summary of Data Model Features

Here's a summary of the features we're talking about having in the data model. These features would be available to parcel developers who are creating new domain-specific schemas, and to end-users who are creating domain-specific schemas.
Most of the features listed below are here because they have some impact on the data model itself. Some of the features may not directly impact the data model, but we've included them here anyway because of their impact on Chandler's end-to-end data handling infrastructure.

Data Model Features

- "in" features -- features we propose to have
  - ✓ multiple inheritance (for kinds)
  - ✓ multi-kind items -- an item can be of more than one kind
  - ✓ item morphing -- an item can change from one kind to another
  - ✓ global attributes -- attributes can be defined globally (like in RDF), or for a kind
  - ✓ compound attributes -- attributes can have component attributes (name: first_name, last_name)
  - ✓ symmetrical relationships between items (but only within a repository)
  - ✓ both weak-reference and strong-reference relationships (but only within a repository)
  - ✓ references across repositories (only unilateral, weak-references)
  - ✓ queries represented as a query data structure (e.g. query items)
  - ✓ both strong-typing and weak-typing for attributes (e.g. "float" vs. "any")
  - ✓ simple garbage collection mechanism similar to ref-counting
  - ✓ unicode string -- all strings in unicode, not ASCII
  - ✓ UUIDs -- 128-bit ids, unique across all repositories
  - ✓ replicated items -- local copies of items from other repositories
  - ✓ enums -- hierarchical enums
  - ✓ schema info as first class items -- items for kinds, types, and attribute definitions

- "future ideas" -- features we might add in the fullness of time
  - 💡 sub-attributes (like in RDF)

- "out" features -- features we want to cut
  - ➡ no symmetrical relationships across repositories

    - no strong-references across repositories
    - no queries that span multiple repositories
    - no queries that span foreign data sources (like IMAP)
    - no text-based query language
    - no versioning -- no DB-level support for item versions

- "open issue" features -- features we haven't decided about
    - strong-typing with multiple types (e.g. "string or Task")
    - unilateral relationships
    - partially replicated items -- proxies for foreign items where the proxy has only some attributes
    - namespaces
    - derivation rules -- on kinds vs. on instances -- indexing derived values -- derived items
    - undo and transactions -- multi-user issues, undo across invocations, etc.
    - permissions -- item-level vs. attribute-level -- capabilities vs. ACLs -- other issues
    - notifications -- conditions, triggers, and notifications
    - files and attachements -- in-repository vs. references to file-system files

---

More detail

- for a more detailed list, see the Data Model Feature List
- and for still more detail, see Data Model Feature Details

Contributors

- BrianDouglasSkinner - 10 Jun 2003

Data Model Feature Details

In May/June 2003, a few of us got together for a series of meetings about what features the data model should have. This document started as a distillation of the meeting notes from those meetings. It may also include some new material, or material from other documents.

The goal for this document is to try to reflect our current thinking, and to describe the data model features in a fair amount of detail.
If you'd rather see a summary document, try one of these:

- Summary of Data Model Features
- Data Model Feature List

Goals

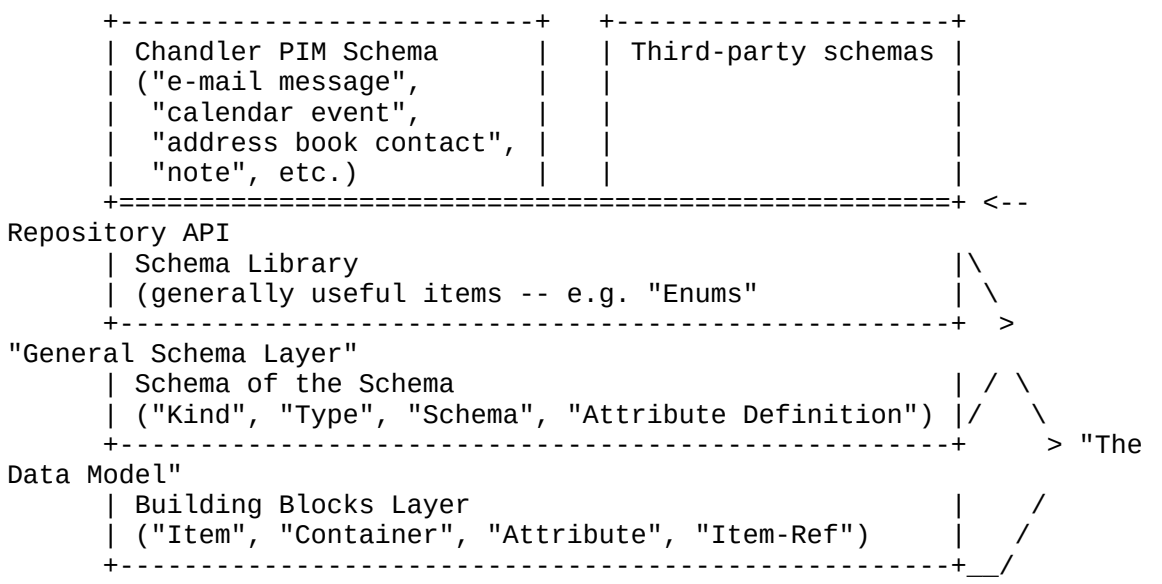
Here are a few of the design goals for the data model:

- **Meet the needs of parcel programmers** -- These are people like the OSAF engineers, who are writing parcels like the OSAF calendar parcel and the OSAF e-mail parcel. Also third-party programmers who are writing their own parcels. Parcel programmers need fixed schemas, strong typing, and guarenteed enforcement of schema restrictions. Parcel programmers are used to thinking about data modeling features from object-oriented programming languages -- inheritance, pointers, etc
- **Meet the needs of end users** -- These are end users who are using Chandlers general purpose info management tools to keep track of unstructured data, and to add structure incrementally. These users need flexible schemas, weak typing, and the ability to easily make ad-hoc changes. These users are used to keeping their records in tools like Excel, or Filemaker Pro, or Access -- which offer slightly different data modeling features.
- **Design something that can be implemented** -- Keep it simple. Don't undertake any research projects.
- **Avoid creating "made up" requirements**

Snakes/Dragons

- Sharing data, sharing schema
- Schema editing, schema evolution
- Query, indexing (we have some rough agreement, still to be worked out)
- Versioning. Recommendation: implement this as an application feature, not a general repository feature.
- Merge/Sync more than two repositories.

Block Diagram



Terminology

We settled on the following terminology:

- reserved words:
  - "Item" -- a bunch of attribute values -- pretty much everything is an item -- e.g. "Lunch with Pat"
  - "Attribute Definition" -- e.g. "Start Time"
  - "Attribute Value" -- e.g. "2:30 pm"
  - "Kind" -- a category of items -- a ***Kind*** has a set of ***Attribute Definitions*** -- e.g. "Calendar Appointment"
  - "Type" -- a type of literal -- "int", "float", "unicode string", etc.
  - "Item-Ref" -- a reference from one Item to another -- e.g. "Employees<-->Department"
  - "Domain Schema" -- a set of ***Kinds*** and global ***Attribute Definitions*** -- e.g. the "Baseball Schema" or the "Chandler PIM Schema"

- non-reserved words:
  - "thing" -- no special meaning in Chandler -- just another fuzzy English language word
  - "schema" -- no special meaning in Chandler -- just another fuzzy English language word

---

Building Blocks

At the ***Building Blocks Layer***:

- Everything is an ***Item***
  - ***Items*** can have ***Attributes***
    - ***Items*** are essentially a dictionary of ***Attributes***
  - all ***Items*** can be accessed through a ***UUID***
    - this is a globally unique id, unique across all repositories
  - all ***Items*** can be accessed through a ***Path***
  - each ***Item*** has a "name"
    - the name may or may not be unique within a ***Container***
    - this is not a user-level display name
- Some ***Items*** are ***Containers***
  - a ***Container*** is just an ***Item*** and has all the aspects of an ***Item***
  - additionally, ***Containers*** contain a bunch of other ***Items***
  - Containership is purely for "house cleaning"
    - Containership DOES NOT IMPLY ANY SEMANTICS
  - ***Containers*** are a facility for the developer, not a facility that the end user need be aware of.
  - **every *Item*** is owned by exactly one ***Container***
  - when a ***Container*** is deleted, all the contained ***Items*** are also deleted
- All semantic references are implemented via ***Item-Refs***
  - ***Items*** never have pointers to each other, except via ***Item-Refs***
  - An ***Item-Ref*** is an object that is not an ***Item***
  - An ***Item-Ref*** can point to an ***Item***, or to another ***Item-Ref***
  - One ***Item-Ref*** connects two ***Items***, via two ***Attributes***, one on each ***Item***
  - ***Item-Refs*** can have policy definitions that dictate how to handle ***Item*** lifecycle events, like what to do when an ***Item*** is deleted or cloned. The policy definitions can dictate when to do deep copies and when to do shallow copies, whether to include sub-items in a delete operation, etc.
- A ***Path*** can be used to identify an ***Item***
  - A ***Path*** is a system level thing, not an end-user thing
  - users would not see ***Paths*** or type in ***Paths***
  - programmers, doing debugging, might type in ***Paths***
- ***Attributes*** can have ***Attribute Values***
  - some ***Attribute Values*** are ***Literals***, like a float or an int or a string
  - some ***Attribute Values*** are ***Item-Refs***
- Conceptually, there are just three things: ***Items***, ***Containers***, and ***Item-Refs*** (plus minutia like ***Attributes***, ***Literals***, etc.)

General Schema Layer

Moving up from the ***Building Blocks Layer***, the rest of this document focuses more on the features of the ***General Schema Layer*** -- the features that will be available to parcel programmers through the Repository API.

The first source of truth for this stuff is the Data Model Feature List. It gives a high-level summary of which features are in and which are out. You should read that document before you read the rest of this one. The rest of this document only has additional notes about the features -- to avoid creating duplicate sources of truth,

this document tries to avoid intentionally including copies of the information from the Data Model Feature List.

Items and Attributes

At the level of the Repository API:

- We will only persist Items and their Attributes, not random python objects
- Every Item has exactly one UUID
- An Item can have attributes, and values for the attributes
- Every Item is essentially a dictionary with key-value pairs of attributes and Attribute Values
- An Attribute Value can be a literal, like the number 14 or the string "Foo"
- An Attribute Value can be an Item-Ref
- An Attribute Value can be null
- An Attribute Value can be a Collection
    - A Collection can contain literals or Item-Refs
    - A Collection can be a List, Set, Dictionary, or Hash

Inheritance

- **four kinds of inheritance**
    - **value**: an instance inheriting a value from another instance -- similar to derived values -- Andi has found this to be an extremely useful feature
    - **attribute**: a ***Kind*** inheriting an ***Attribute Definition*** from a "superclass" ***Kind***
    - **type**: for strongly typed schemas, when type-checking, an instance counts as type Foo either when it is an instance of the Foo Kind, or when it is an instance of a Kind which is a "subclass" of the Foo Kind
    - **behavior**: in Python or Java, classes inherit methods from superclasss -- in a database, records do not have behavior -- in our Python mapping for the data model, we will map Python classes to Kinds in the database
- we resolved
    - to support **attribute** inheritance
    - to support **type** inheritance
    - to support multiple inheritance (and therefore also single inheritance)
        - but, to try not to use multiple inheritance in the Chandler PIM Schema
    - to allow an item to be an instance of two kinds
        - but, in the Chandler parcels, not to offer any kind of general data mixing features, but rather in the Chandler PIM Schema, to handle as the E-mail/Task mix as a special case
        - see the dev list discussions with Micah Dubinko:
            - http://lists.osafoundation.org/pipermail/dev/2003-May/000662.html
            - http://lists.osafoundation.org/pipermail/dev/2003-May/000665.html

    - the "Building Blocks" layer will provide the foundation for implementing **value** inheritance -- we want to talk more about value iinheritance later, when we're also talking about derived values in general -- then we need to decide what support we want to offer for **value** inheritance up at the level of the "Repository API"

- **"Emergent" typing for kinds** vs. **Declarative typing for kinds**
    - see the dev list discussions with Micah Dubinko:
        - http://lists.osafoundation.org/pipermail/dev/2003-May/000662.html
        - http://lists.osafoundation.org/pipermail/dev/2003-May/000665.html
        - http://lists.osafoundation.org/pipermail/dev/2003-May/000666.html
    - resolved to pick declarative typing for kinds
    - we won't provide any direct support for "emergent" typing, although a third-party parcel developer could write a parcel that had this feature

---

Attributes

- **end user** vs. **parcel programmer**
    - The end user wants malleable data, the ability to construct arbitrary views and do interesting searches. The programmer wants objects to use when writing code: some reliance on structure, easy persistence, etc. We've struggled with these two design goals, we've occasionally captured this tension as "item centric" vs "object centric". We could also refer to these two things as the "data model" and the "object system" respectively.
    - It may be a good idea to formally separate attributes (on a given item) into end-user-visible-attributes and programmer-housekeeping-attributes, where there are different expectations for the different types.

- **"domain attributes" vs. "house-keeping attributes"**
    - any given item will have both domain attributes and house-keeping attributes
    - domain attributes
        - domain attributes are things that the end-user cares about, like a baseball player's "name", or "age", or "batting average"
        - domain attributes should always be visible to the user
    - house-keeping attributes
        - house-keeping attributes are things that the chandler infrastructure code cares about, like "last-modified" time, or "version" number, or a "logically deleted" flag
        - house-keeping attributes may frequently be invisible to the user, although in some cases the user might want to be able to look at them (e.g. "creation date")
    - probably users should never be able to edit house-keeping attributes directly

- **"display names" and "identifier names"**
  - an attribute definition can have an optional identifier name -- e.g. "startTime"
    - the identifier name syntax restrictions that would work in Python or Java
    - the identifier names must be unique within a Kind
  - an attribute definition can have an optional display name, which is what the end-user sees -- e.g. "Start Time"
    - the display names are not required to be unique within a Kind -- we want to be flexible, even though we can imagine this flexibility will make for some headaches later (e.g. in some future text-based query language)
  - an attribute definition might also have a URL that uniquely identified it across namespaces, although this might be unnecessary because an attribute definition can always be uniquely identified by its UUID

- **"global attributes" vs. "local attributes"**
  - see also: DataModelIssues#Global_Attributes
  - an Attribute Definition can be a "global attribute"
    - a global attribute can be used by more than one Kind
    - a global attribute is defined in a Domain Schema
    - a global attribute can be used by Kinds from other Domain Schemas
  - an Attribute Definition can be a "local attribute"
    - a local attribute is defined within a single Kind, and used only by that Kind

- **"sub-attributes"**
  - see also: DataModelIssues#Sub_Attributes
  - we are not going to support sub-attributes for 1.0
  - this should be something that could be added after 1.0 without breaking anything (although we did note that it might be difficult to write database code that would be efficient when processing queries on a super-attribute)

- **Domain Schema Diagram**
  - We settled on diagram showing how the schema info is organized. I don't have a good way to reproduce the diagram here, but here's what it shows:
    - A ***Domain Schema*** item has a collection of ***Kind*** items
    - A ***Domain Schema*** item has a collection of ***Attribute Definition*** items, representing global attributes
    - A particular data item (e.g. "Lunch with Pat") has a defining ***Kind*** item
    - A ***Kind*** item has a collection of ***Attribute Definition*** items
      - some of those ***Attribute Definition*** items may be local to this ***Kind*** item
      - some of those ***Attribute Definition*** items may be global attributes that are in the collection of ***Attribute Definition*** items pointed to by the ***Domain Schema*** item that is pointed to by the ***Kind*** item

            - some of those ***Attribute Definition*** items may be "imported" global attributes that are in the collection of ***Attribute Definition*** items pointed to by some unrelated ***Domain Schema*** item
        - An ***Attribute Definition*** item may be pointed to by more than one ***Kind*** item
        - An ***Attribute Definition*** item may be pointed to by at most one ***Domain Schema*** item

- **Attribute Definitions** vs. **Attribute Bindings**
    - When a ***Kind*** item includes an ***Attribute Definition*** item, the ***Kind*** item uses all the general information defined in the ***Attribute Definition*** item, which is shared by all the ***Kind*** items that use the ***Attribute Definition*** item
    - In addition, there may be some specific information particular to the use of the ***Attribute Definition*** item in this specific ***Kind*** item -- information that not's associated with the ***Attribute Definition*** item itself, but with the binding of the ***Attribute Definition*** item to the ***Kind*** item
    - Here's a breakdown of what we decided about what information should be associated with the ***Attribute Definition*** item and what should be associated with the binding
        - Attribute Definition info
            - "type"
                - could be something like int, float, string, date...
                - could be a specific sort of Item-Ref
                - could be "Any", meaning any of the above
            - "one vs. many"
                - this is really "cardinality" info -- but we want to be clear that we're only offering the two choices, "one" and "many", rather than more complicated things like "4" or "6 to 8"
                - defaults to "many" when a "baseball fan" creates a new Attribute Definition
            - "identifier name"
                - used as a python token -- e.g. "startTime"
            - "display name"
                - appears in the UI -- e.g. "Start Time"
                - can be a simple ASCII string, or a Unicode string, or a "Polyglot string" (meaning a dictionary of localized string translations, keyed by langauge)
        - Attribute Binding info
            - "required"
                - a boolean value -- means the same as "not null" -- the attribute must be included in every instance, and must always have a value
    - There are a few different options for storing Attribute Binding info:
        - We could have separate Attribute Binding items between a ***Kind*** item and an ***Attribute Definition*** item
        - We could somehow associate it with the Item-Ref that relates a ***Kind*** item to an ***Attribute Definition*** item

        - We could somehow associate it with the the (attribute of the ***Kind*** item) that points to the Item-Ref that points to the ***Attribute Definition*** item
            - We might be able to do this just by using our "Compound Attribute" idea
        - We didn't pick which option we want -- we'll cross that bridge when we come to it

- **attributes on attributes**
    - example: Spot is a dog. Spot has an attribute called "age". Spot's age is "6". Can "Spot's age is 6" have other attributes, that further describe the "age" of Spot? Maybe attributes like "percent certainty", or "error term" or something?
    - resolution: no, we're not going to support this

Relationships

- **Symmetrical Associations**
    - (see also: DataModelIssues#Symmetrically_Associated_Attribu)
    - we will support Symmetrical Associations within a repository, via Item-Refs

- **Item-Refs**
    - we will have Item-Ref instances (between two Items)
    - we will not have "Item-Ref Definitions" that the Item-Ref instances point to
    - End users can create ad-hoc relationships between Items using Item-Refs

- **strong references and weak references**
    - we will offer support for both "strong references" and "weak references"
    - "strong references" and "weak references" can be implemented via Item-Ref house-keeping policies like "don't delete on delete"
    - "strong references" will be used to do garbage collection using something similar to reference counting

- **referential integrity**
    - The repository will ensure referential integrity, but only for Item-Ref relationships
    - When you delete the item at one end of an Item-Ref, the Item-Ref **always** goes away

- **cycles**
    - We agreed that its ok to have cycles in the schema, the repository will be able to handle it.
    - see the dev list discussions with Micah Dubinko:
        - http://lists.osafoundation.org/pipermail/dev/2003-May/000662.html

        - http://lists.osafoundation.org/pipermail/dev/2003-May/000665.html
        - http://lists.osafoundation.org/pipermail/dev/2003-May/000666.html

- **one-to-many and many-to-many relationships**
    - we will have support for
        - one-to-one relationships -- via attributes that point to Item-Refs
        - one-to-many relationships -- via attributes that point to Lists of Item-Refs
        - many-to-many relationships -- via attributes that point to Lists of Item-Refs that point to Lists

- **unilateral relationships**
    - use case where you might want this: an item needs to know its kind, but a kind doesn't want to know about every item.
    - instead of a full ***ItemRef***, the item could reference the kind via the ***uuid***
    - the kind/item is a special case (fundamental because it is so common), but similar desired functionality could be required in other cases
    - should a "unilateral relationship" be a first class concept in our data model?
        - pro: would be useful to specify unidirectional relationship and say something about the domain/range of the relationship. Useful for introspection. Useful for debugging. Possibly useful for constraint enforcement. Without it, you can't have a self describing schema of schemas
        - con: proliferation of first class concepts complicates the repository. Also, more difficult to enforce range or other restrictions on a unilateral relationship.
    - resolved to leave this as an open issue for now.

- **remote relationships**
    - How do we handle strong references across repositories?
    - questions:
        - do item-refs work across repositories?
        - does garbage collection work across repositories?
        - how can items in different repositories be related?
    - answers:
        - trying to get item-refs and garbage collection to work across repositories constitutes a research problem, which we want to avoid.
        - which should only have simple, low-tech mechanisms for cross-repository relationships
    - how do we refer to a repository?
        - we might have local repository items that represent remote repositories
        - items might refer to a repository by the repository items, or by having a url that points to the remote repository
    - resolved:

- we will have a separate data type to represent an "external reference" -- an external reference is a weak reference to an item in a remote repository
- when you subscribe to an item instance in a remote repository, you get a local copy of the remote item
- you can view items in other people's repositories, but if you're just viewing (not subscribing) then those items never get stored in your repository

---

Compound Attributes

(see also: DataModelIssues#Compound_Attributes)

- a compound attribute will just be a normal Attribute Definition, but one that has a list of other component ***Attribute Definitions***
- the compound attribute will have all the features of a normal Attribute Definition, like "identifier name", and "display name", and "one vs. many" cardinality
- the component-attributes will each have all the features of a normal Attribute Definition, like "identifier name", and "display name", and "one vs. many" cardinality

---

Indexing

- **indexes** for **queries**
  - in a ***Domain Schema***, Attribute Definitions can be explicitly marked with a flag to declare that they should be indexed (the ***Domain Schema*** is in a layer above the ***Building Blocks***)
  - we could imagine having the repository try to observe queries and heuristically index, but that would be a research problem, so we're not going to do that
  - full text indexes are a special case. Full text indexes will live with the data. We hope to use an existing 3rd party one, but may run into problems, especially getting it to work well with transactions. John's confident it wouldn't be **too** hard to roll our own if we needed to.
  - when a transaction fails (or on rollback), the index needs to be rolled back
  - when an item is deleted, the entries for it need to be removed from the index

---

Queries

- **goals**
  - searches of structured data should be fast
  - searches of free-form data should be possible

- **text-based query langauge**
  - we need to support some kind of programmatic API for submitting a query -- e.g. using some data structure to represent a query
  - we don't want to have a text-based query language that requires parsing

- **joins**
  - Unlike SQL, we're not doing "joins"

- **query sorting**
  - A query may also need to specify some sort of grouping/sorting behavior. Queries don't just return result sets, they return the result set pre-sorted, so that client code can get just a partial result set and then start interating over the entire result set as needed by the UI.

- **query contexts**
  - For a query, we often want specify the set of things to look through, and a filter on that set. The key is to narrow the set to be smaller than the whole repository.
    - Kind-specific queries
      - Most queries/searches are done in some context, and you can usually limit the set based on ***kinds***. For example, a query might look for all the e-mail from a given person, so then we only have to look at items of ***kind*** e-mail, not all items. And calendar views generally only look at calendar events.
    - Multi-kind queries
      - In some cases, queries span different kinds. A calendar view might be used to view e-mail messages. And a generic table view might be used to look at lots of different kinds of items side by side.
    - Owners
      - The notion of an "owner" seems important for limiting queries. Many items will be owned by one user -- e.g. I own my e-mail. For lots of common queries a user will just want to search through stuff they own -- e.g. I mostly just want to see my calendar, my e-mail, my tasks.
  - Most queries will only need to search over the set of ***InformationItems***. ***InformationItems*** are things like Email, contacts, appointments, tasks, notes. The set of ***Items*** is more general, and might include "system info" items, like items representing parcels, agents, repositories, queries, etc.

Values

(see also: DataModelIssues#Null_Values)

Default Values

(see also: DataModelIssues#Default_Attribute_Values)

Restrictions

(see also: DataModelIssues#Restrictions_on_Attributes)

- A schema designer can place three types of restrictions on attribute values:
  - Cardinality restrictions
  - Requried (non-null) restrictions
  - Data-type restrictions

- Cardinality restrictions
  - attributes can have cardinalities of **One** or **Many**

- Required/optional
  - attributes can be either **required** or **optional**

- Strong typing
  - Questions:
    - Can we rely on a "startDate" to always be a date, not just some random text string?
    - Is the user restricted from entering invalid dates? Do we do data entry validation?
    - Does the repository do type checking when it gets a request to store a value?
    - Does some layer of code uniformly do type checking whenever a value is retrieved from the repository?
    - Can a GUI parcel programmer trust that something is what it's supposed to be, or does that person have to do defensive coding everywhere?
    - Should there be some code that does a repository validation sweep, traversing the repository and ensuring that it is valid?
  - Answers:
    - A schema can specify that some ***Attributes*** are strongly typed
    - A schema can specify that some ***Attributes*** are not strongly typed
    - For strongly typed ***Attributes***, we build the plumbing such that the GUI parcel programmer can blindly trust that the ***Attribute*** will never get a value of the wrong type
    - For ***Attributes*** that are not strongly typed, an end user is free to enter whatever value they want (e.g. via an tool like the SuperWidget)
    - For ***Attributes*** that are not strongly typed, when the ***Attribute Value*** is stored in the repository, we have to also store something telling us what "type" the value is (e.g. if we store xF800, we need to know whether it's a int or a string)

Item Versions

(see also: DataModelIssues#Item_Versions)

- **item versions**
  - this is a snake/dragon
  - what would it mean to have garbage collection in a system with change logs and item histories?
  - what would it mean to have an explicit delete operation in a system with change logs and item histories?
  - supporting item versions in a general way could end up being a research project
  - resolution: we recommend that any item version features should be implemented on a case-by-case basis up in the application parcel code, rather than having the repository offer any kind of ubiquitous general-purpose version mechanism for all items

---

Deletion

(see also: DataModelIssues#Garbage_Collection)

- **explicit delete**
  - in addition to garbage collection, we will also provide for an explicit delete operation
  - it will be possible to associate different deletion semantics with different relationships, by setting the "policy bits" on Item-Refs
  - once you delete an item it is gone forever

Strings

(see also: DataModelIssues#Text_Strings)

- **strings**
  - Unicode everywhere at the user level, not ascii
  - symbols -- used for python mapping -- ASCII strings with no spaces, etc.
  - Polyglot strings, most likely -- (dictionaries of translations, keyed by language)
  - ***any string***, possibly -- (an attribute of type ***any string*** could have a value that was a unicode string, or a polyglot string, or a rich text string
  - more complex types of strings are allowed (all can be represented by ascii), but captured at a higher level (building blocks is unaware of them)
  - rich text strings should be represented in some platform independent way, rather than storing the different platform-specific rich text strings that one encounters on Linux and Windows and Mac

---

Schema Items

- **attribute definitions**
  - we will represent attribute definitions as Items
  - for example, there will be a "Start Time" Item, of Kind "Attribute Definition"
- **kinds**
  - we will represent kinds as Items
  - for example, there will be a "Contact" Item, of Kind "Kind"
- **types**
  - We will try and represent each Type as an Item
  - for example, there will be an "Int" Item, of Kind "Type"
  - 3rd parties being able to define their own types is a possible later feature, will not be addressed out of the gate.

Basic Data Types

- Can parcel developers define new ***Literals***?
  - maybe

RDF

- The Chandler data model differs from the RDF model.
- We do not aim to be a general triple store, capable of storing arbitrary RDF
- We will not be able to import arbitrary RDF -- only RDF that we have exported
- We can easily write some code to export data from a Chandler repository to some kind of RDF document, as long as we get to pick our own RDF representation. We want to be able to export to some sort of RDF, but it can be a general Chandler-centric RDF, not an RDF designed for interoperation with other apps in specific application domains.
- We haven't yet identified other specific ways that we might want to interoperate with RDF, and we don't know how well-suited the Chandler data model will be for interoperating with RDF tools.
- Chandler application level parcels may want to have their own RDF import/export tools, to express their data in a particular rdf schema (calendar, foaf, dublin core)

UUIDS

(see also: DataModelIssues#Unique_IDs)

- **UUIDS**
  - UUIDs will be randomly generated 128-bit values
  - different types of datastores will not each assign their own flavor of UUID

    - the UUID will not be a tuple that includes (or encodes) the "home repository" of an item
    - the client be able to assign its own UUIDs, without having to request them from the server?
    - the kind item for "Contact" will have the same UUID in every repository

---

Annotations

Foreign Items

(see also: DataModelIssues#Local_Proxies)

- **foreign items**
    - we will definitely want to have local copies of items from foriegn data sources (like IMAP mail), in order to handle indexing and querying from within Chandler, and in order to work when off-line
    - we will definitely want to have local copies of items from other Chandler repositories, for example when a user has subscribed to something from somebody else's repository
    - resolutions:
        - this problem will probably end up belonging to whoever is doing replication -- we're going to leave it be for now

---

Enums

(see also: DataModelIssues#Enumerated_Types)

- there are a couple different options for providing enum support
- we resolved:
    - the "Building Blocks Layer" should have support for "symbols"
    - the mechanisms in the "Building Blocks Layer" will probably be sufficient to support whatever type of enum representation we used in the "General Schema Layer"

Namespaces

(see also: DataModelIssues#Namespaces)

- Do we want the repository to have some notion of namespaces, or are namespaces something that should be handled by semantic dictionaries?

- **namespaces**

    - we might not need namespaces, because items can always be refered to by UUID rather than by name
    - resolution: left this as an open issue for now

Derivation Rules

(see also: DataModelIssues#Derived_Attributes)

- Can we live with not having derivation rules associated with individual attributes of individual items, and instead just have derivation rules associated with attribute definitions at the class level?
- Can we say that derived values always calculated by the client?
- Can we say that derived values are stored in the repository?

Repositories

(see also: DataModelIssues#Repository_Subsets)

Display Strings

(see also: DataModelIssues#Display_Strings)

Transactions

Permissions

- **authorization** and **permissions**
    - once we hire a security person then we'll take direction from them about how our overall authorization system should be designed
    - in any case, authorization needs to be enforced on the server side -- code on the client never even gets to see any items or attributes that the user doesn't have permission to see
    - therefore, the database will need to know about the permissions schema in order to enforce the permissions
    - authorization might be handled using "capabilities" or "ACLs"
    - resolved: come back to this once we have a security person

---

Parcels

Notifications

Sharing

Files

Schema Changes

- **background**
  - **parcel programmers** may make changes to the schema, but the changes will be carefully planned, and the new parcel code will be written so that it (a) can do data conversion on existing items, and (b) can deal with running into old-format data
  - **end users** will also change the schema information, by doing things like adding attributes to a kind, or deleting attributes, or changing the names of attributes. Those changes will impact items that already exist. But end users won't write data conversion routines, so Chandler will have to deal gracefully with that.
  - in both types of use case, there will be some thorny issues. once we have a data model design that describes what schemas look like, then we need to cycle back and think about what types of changes we will allow people to make to their schemas, and what we need to do to support those schema changes.

- **merging items**
  - can you take two items that different UUIDs and merge them so that you end up with only a single item?
  - might be desirable from a user's point of view, but implementing this in a general way would be hard

- **schema editing**
  - can a user edit the Chandler PIM schema?
    - example: Joe adds a new "hair color" attribute definition to the "Contact" kind
    - resolution: NO, all the PIM Schema instances are read-only -- more generally, any schema instance should be read-only if there's ever python code written against it
  - can a user create their own sub-kind of "Contact" and add attributes to that?
    - creates lots of complexity when users start sharing items that come from schemas that were originally related but have been forked

---

Contributors

- BrianDouglasSkinner - 10 Jun 2003

Menu of possible data model features

This document offers list of possible features we might want to have in the data model. These features would be available to parcel developers who are creating new domain-specific schemas, and to end-users who are creating domain-specific schemas.

As of June 2003, we're in the process of going through this list. We're picking which features we think are important, which we think we can discard completely, and which we might think about adding someday in the future. Once we've finished going through the list, we plan to summarize our suggestions in a proposal.
Once we have a complete proposal, we'd like to have some kind of review for that proposal. The review should eventually result in "truth" -- meaning we will have signed off on a feature list that says what data-representation features Chandler will have. That feature list will end up impacting the design of lots of different parts of Chandler: the database code, the repositry API, the query langauge, RAP, the Python-binding code for the data model, the Chandler PIM schema, etc.
This is just a bullet-point list of features -- for more background info, and more detailed discussion about the features, see:

- Data Model Feature Details
- Data Model Issues

Our recommendations are marked below:

- ✓ means "in" -- We propose that Chandler should support this feature.
- ➡ means "out" -- We propose that the Chandler platform itself should not have first-class support for this feature (although in many cases third-party parcels might be able to offer this feature, built on top of Chandler building blocks)
- 💡 means "idea for future version" -- We think this might be a good idea for later. We propose not to include this for Chandler 1.0, but to keep it in mind as a possible later extension.
- ? means "open question" -- we haven't yet made an in/out recommendation

---

menu

- **kind inheritance** -- (see 29 May 2003 meeting and 30 May 2003 meeting)
  - single inheritance vs. multiple inheritance
    - ➡ no kind inheritance
    - ➡ just single kind inheritance
    - ✓ multiple kind inheritance
  - items as instances of a kind
    - ➡ an item can be an instance of just one kind
    - ✓ an item can be an instance of one or more kinds
      - ✓ declarative typing for kinds
      - ➡ "emergent" typing for kinds

        - an item can change from one kind to another
            - for the case of a kind that does not have associated Python parcel code
            - for the case where there is Python parcel code (e.g. Calendar Event or E-mail Message), but in carefully restricted ways, not willy-nilly however the user wants to
    - aspects of inheritance
        - "attribute" inheritance -- a sub-kind inherits attribute definitions from the super-kind
        - "type" inheritance -- an instance of a sub-kind counts as being of the type of the super-kind
        - "value" inheritance -- the value of one attribute is derived from another
        - "behavior" inheritance -- sub-kinds inherit functions or methods
            - data model knows about behavior
            - behavior stored only in Python classes which are mapped to Kinds in the schema

- **attributes** -- (see 30 May 2003 meeting & 02 Jun 2003 meeting)
    - global attributes
    - attributes bound to a kind
    - ad-hoc attributes on an instance
    - sub-attributes
    - attributes used to describe items
    - attributes used to describe other attributes
    - attributes flagged as ones to index on -- (see 04 Jun 2003 meeting)

- **relationships**
    - unilateral relationships -- (see 04 Jun 2003 meeting)
    - symmetrical relationships -- (see 28 May 2003 meeting)
    - one-to-one relationships -- (see 04 Jun 2003 meeting)
    - one-to-many relationships -- (see 04 Jun 2003 meeting)
    - many-to-many relationships -- (see 04 Jun 2003 meeting)
    - weak-reference relationships -- (see 02 Jun 2003 meeting)
    - strong-reference relationships -- (see 28 May 2003 meeting)
    - inter-repository relationships -- (see 04 Jun 2003 meeting)
    - circular references -- (see 04 Jun 2003 meeting)

- **compound attributes (wholly-owned sub-items)** -- (see 02 Jun 2003 meeting)
    - compound attribute definitions as first-class items
    - component attribute definitions as first-class items
    - component attributes can be Item-Refs
    - queries can refer to component attribute definitions
    - ad-hoc component attributes

    - component attributes can be required or optional
    - component attributes can have cardinality "many"

- **queries** -- (see 04 Jun 2003 meeting)
    - one-off queries
    - ongoing queries
    - persistent queries
    - observable queries
    - queries spanning mutiple repositories
    - queries spanning foriegn data stores

- **values** -- (see 28 May 2003 meeting)
    - null values
    - missing attributes

- **default values**
    - no default value support
    - default stored on attribute definition
    - default values are set when item is created
    - default values are set when attribute is created
    - on get(), default values are returned but not set

- **restrictions** -- (see 28 May 2003 meeting)
    - cardinality restrictions
    - non-null restrictions
    - data type restrictions (e.g. "int")
        - strong typing available
        - weak typing available
    - multiple type options (e.g. "int or Event") <-- probably no; but maybe if it's easy; talk to Andi
    - range restrictions (e.g. "21 to 35")
    - inter-attribute restrictions
    - transitive restrictions
    - "expectations" vs. restrictions

- **item versions** -- (see 05 Jun 2003 meeting)
    - any versioning features are implemented by custom app code
    - built in database support for versioning
        - past versions of items
        - change objects
        - current version number in items
        - version tumblers

- **deletion** -- (see 05 Jun 2003 meeting)
    - explicit deletion
    - explicit deletion of current version

    - explicit deletion of all past versions
    - simple garbage collection similar to ref-counting

- **strings** -- (see 05 Jun 2003 meeting)
    - symbols (ASCII, no spaces, etc.) -- (available at system level, but not user level)
    - ASCII strings -- no, store as byte array
    - 16-bit unicode strings
    - 32-bit unicode strings
    - platform independent rich text strings
    - platform specific rich text strings
    - HTML links in strings
    - Chandler links in strings
    - polyglot strings -- dictionary of localizations keyed by language
    - any string -- matches any of the above

- **meta-data as items**
    - kinds
    - attribute definitions
    - types
        - aliases for types
        - third-party parcels can create new types

- **basic data types**
    - data types geared towards end users and third-party schema authors
        - int, float, boolean, etc.
        - List
        - Item-Ref
        - cross-repository reference
        - blob
        - date, time, duration, timespan, timezone
        - URL, file
    - data types geared towards parcel authors
        - symbol
        - UUID
        - repository containment path
        - third-party developers can add basic data types

- **import/export** -- (see 05 Jun 2003 meeting)
    - domain specific RDF formats (e.g. Calendar)
    - mappings to other RDF schemas
    - Chandler-centric RDF format (bare-bones, and PIM ignorant)
        - RDF import
        - RDF export

- **uids**
    - tuples (repository, local-id)

    - non-tuple ids (uid)
    - homogenious uids
    - heterogenious, store-dependent uids
    - client assigns it's own uids
    - only servers assign uids

- **annotations**
    - item-level annotations
    - attribute-level annotations
    - simple text-appending annotations
    - complicated editing annotations

- **foreign items** -- (see 04 Jun 2003 meeting)
    - local copies from other Chandler repositories
    - local copies from foriegn data sources (e.g. IMAP)
    - full proxies
    - partial proxy copies

- **enums**
    - no support
    - single-level enums
    - hierarchical enums
    - ordered enums
    - enums with multiple orderings
    - enums implemented as items
    - enums implemented as symbols

- **namespaces**
    - namespaces for attributes
    - namespaces for items
    - semantic dictionaries for mappings

- **derivation rules**
    - derived attributes on a kind
    - derived attributes on an instance
    - derived values indexed on the server
    - circular derivations
    - derived items (not derived attributes)

- **users**
    - users
    - personas
    - profiles
    - groups
    - contacts
    - agents

- **repositories**
  - repository subsets
  - virtual repositories

- **display strings**
  - standard display string attribute
  - kind-specific display string attributes
  - derived display strings -- e.g. value inheritance

- **transactions**
  - rollback
  - undo
    - single-user undo
    - multi-user undo
    - intra-session undo
    - inter-session undo

- **permissions**
  - item-level permissions -- (see 04 Jun 2003 meeting)
  - partial-item permissions
  - group permissions
  - kind-instantiation permissions
  - domain attribute vs. house-keeping attribute permissions

- **parcels**
  - installing parcels via sharing

- **notifications**
  - conditions
  - triggers
  - notifications

- **sharing**
  - publishing
  - subscription

- **files**
  - attachements
  - documents
  - in-repository storage
  - file-system storage

Contributors

- BrianDouglasSkinner - 21 May 2003 to 11 Jun 2003

Data Model -- Issues

This page discusses all the issues that have come up in thinking about the Chandler data model.
You may also want to look at the other data model pages:

- **Data Model**
  - **Data Model Introduction**
  - **Data Model Issues** <-- you are here
  - **Data Model Schema**

Contents

(all of these links stay inside this page)

Overview

This document has a list of 20 (or so) discussion topics about that data model design. There are probably a few more topics that should be in that list, plus some of the existing topics should be fleshed out a little more.

Each of those discussion topics raises one or more issues, presenting open questions and sometimes suggesting options or alternatives. Eventually we'll need to resolve those issues. The issues vary a good deal in how complicated they are, and they also vary a good deal in when they need to be resolved -- which ones have impact now and which ones can wait.

In the long run, for each issue we might want to follow some design process, with steps like these:

- discuss the issue
- propose solutions
- identify dependencies
    - identify dependencies between the data model issue and other parts of the Chandler architecture
    - for example: if we want to support some feature Foo in the data model, then the data store must have support for Foo-tabs, or at least Foo-proxies.
    - in practice, a lot of the data model issues may depend on broader design decisions, like what features end up getting implemented in the client vs. the repository vs. the middleware
- decide if this is needed now
    - identify features that are needed soon
    - identify features that need to be designed for now, to avoid painting ourselves into a corner
    - identify features that could easily be added in later, and don't need to be considered now
- pick a solution
    - document the fact that we've picked the solution

But in the short run we might just want to focus on the most important issues -- the issues that have the broadest impact, or that need to be resolved first. Here's a first pass at what I think might be the "top 10" issues to try to get resolved first

1. **Terminology**
    - Sign off on a core set of terms. Maybe just the first couple dozen most imporant ones, like Thing and Class and Kind-of-Item.
2. **Relationships**
    - Do we want to support symmetrical associations? One-to-One? One-to-Many? Many-to-Many?
    - Do we also have support for non-symmetrical associations?
3. **Garbage Collection**
    - Does the data model provide the notions of both strong and weak references? Or do we say that everything is a strong reference?
    - How do we handle strong references across repositories?
    - What does it mean to have garbage collection in a system with change logs and item histories?

    - Do we provide for an explicit delete operation, and what are the semantics of that across relationships? Are different deletion semantics associated with different relationships?
4. **Global Attributes**
    - Are attribute definitions always associated with particular classes, or are they always global, or can they be either global or associated with a class?
    - Are global attributes truly global, or are they defined within some scope (e.g. within a parcel)?
5. **Derived Attributes**
    - Can we live with **not** having derivation rules associated with individual attributes of individual items, and instead just have derivation rules associated with attribute definitions at the class level?
    - Can we say that derived values always calculated by the client?
    - Can we say that derived values are stored in the repository?
6. **Namespaces**
    - Do we want the repository to have some notion of namespaces, or are namespaces something that should be handled by semantic dictionaries?
7. **Restrictions**
    - Should we provide for **any** restrictions on attribute values? Cardinality? Requried (non-null)? Data-type restrictions?
    - If we have restrictions, where are they enforced?
    - If we have restrictions, can they be overridden?
    - Should we have "expectations" instead?

To Do List

This strawman data model is already a long document, but there's still lots of work left to do. Right now I (Brian) don't even have a presentable first draft. Here's a list of pending issues:

- Think about how to have any (item with a date) appear in week view
- Search for "@@@". For each of the dozens of instances, spend an hour writing the discussion that needs to go there.
- Spell-check this whole thing

Terminology

In building Chandler, OSAF is using Python, and RDF, and other software components and data standards. Each of these things has its own set of terminology. Often the terminology overlaps. There are "classes" in Python, and there are "classes" in RDF, but a Python class isn't the same thing as an RDF class. Chandler introduces its own set of terms: like "Parcel" and "Item" and "View". Some of these terms also overlap with existing terms in Python and RDF and other building blocks.

In the course of making this strawman data model, I've had the opportunity to think about what to name all the different things in the data model. Ideally, it'd be good if the names were short, descriptive, and unambiguous. Those goals are in tension. You can make a name really short, or you can make it really descriptive, but if you try to make it short **and** descriptive you end up making some trade-offs on both ends.

The goal of being unambiguous is also in tension with the goals of being short and descriptive. We could say each "item" in a Chandler item store is a member of a "class" of items, but then when you talk about a "class" it's not clear whether you mean a Python class, or a Chandler class, or an RDF class.

One solution would be to always say "Chandler class" when you're talking about a Chandler class. But it's not realistic to expect people to always fully qualify their language that way. Another solution is to use some short prefix for the Chandler terms, so that we'd talk in terms of "OsafClasses" and "OsafAttribute". And then the word "class" always means a Python class or an RDF class, not an OsafClass.

Yet another solution is to try to pick Chandler terms that tend not to overlap too badly with Python terms and RDF terms. For this first pass at a data model, that's the approach I've tried to take. Unfortunately, one of the side effects is the terms can be a little cryptic. These new terms may take some getting used to. And they may just be a bad idea. If people don't like them, then we can pick a different solution and just do a few global replaces in this file. In any case, I hope the terminology won't get in the way of the ideas.

Typical terminology from an object oriented programming language	Examples
"class" or "type"	Dog, Cat, Animal
"object" or "instance"	Rover the Dog
"attribute" or "property" or "instance variable" or "member variable"	Rover.birthday
"instance of" or "isa" or "type"	"Rover is an instance of a Dog" or "Rover isa Dog" or "Rover is of type Dog"
"subclass of" or "a kind of" of "ako" or "isa"	"Dog is a kind of Animal" or "Dog is a subclass of Animal"
"superclass" or "parent class"	"Animal is the superclass of Dog" or "Animal is the parent class of Dog"
"literal type" or "type"	string, int, float

- Python Terminology
  - class
  - type
  - property
  - attribute
  - instance
  - object

- RDF Terminology
  - (including terminology from XML, XSD, RDFS, and DAML+OIL)
  - element

    - attribute
    - attribute name
    - attribute value
    - triple
    - Property
    - predicate
    - sub-property
    - Class
    - sub-class
    - Datatype
    - Individual
    - Object
    - type
    - value

- Database Terminology
    - "Database Management System (DBMS)" or "Datastore"
    - "Database" or "Datastore"
    - Table
    - Record
    - Column
    - Field

Possible Chandler Terminology		
name	**examples**	**description**
Kind of Thing	Person, Place	Like a class in an OOPL, or a table in a RDBMS
Kind of Item	Event, Email, Contact	a sublcass of Thing representing "information items" -- tends to have a 'body' attribute, and is the major unit in the Chandler PIM
item	the "Tom Jones" Contact item	An instance of some Kind-of-Item. A collection of attribute values, with a uid
Attribute Definition	"due date"	Meta information about an attribute. Includes type, cardinality, etc.
Attribute Value	"6 Oct 2008"	The value of a particular attribute for a particular Thing
instance of	instance of a Dog"	

Data Model Requirements

Here's a list of the different requirements I had in mind as I was working on this strawman data model. As we keep working we can add or remove requirements here, and we can try to better flesh out the details about exactly what these requirements mean.

We want to avoid coming up with "made up" requirements. If some part of Chandler **needs** a feature to be supported by the data model, then that should become a

requirement for the data model. Or, in some cases, we may anticipate a future need, and recognize that if we ever want to support that need, then we have to design for it from the beginning. But where possible, we should try to defer adding features until some later date, when or if they are really needed.

- High Level Requirements
  - A user can store relatively unstructured data.
  - A user can store structured data.
  - A user can add structure to unstructured data.
  - A user can create complex queries across a sea of data.

- Possible Basic Requirements
  - A user can create a new Item that is an instance of an existing Kind-of-Item.
  - A user can create a new Item that isn't of any specified Kind-of-Item.
  - A user can assign an existing Item to be a different Kind-of-Item.
  - A user can create a new Kind-of-Item.
  - A user can create a new attribute for any Kind-of-Item.
  - A user can create a new attribute for any existing Item.
  - A user can change the name of an existing attribute, or modify it in other ways.
  - A user can restrict the range of an existing attribute.
  - A user can delete an Item.
  - A user can delete a Kind-of-Item.
  - A user cannot delete basic system Items that are needed to bootstrap the system.
  - A user cannot cause the data model to become corrupt or invalid.
  - A user can view an old version of an Item.
  - Code in a new parcel can do anything a user can do.

- Specific Requirements
  - Support for enumerated types (ENUMs)
  - Support for hierarchical ENUMS
  - Support for derived attributes

- Possible Further Requirments
  - Subclasses
    - A Kind-of-Item can be a subclass of another Kind-of-Item.
  - Sub-properties
    - An attribute can be a sub-property of another attribute
  - NULL values for attributes
  - Display Strings for Items
  - Default Values for attributes
  - Namespaces
  - Restrictions on attribute values
    - required/optional
    - cardinality
    - data type
    - complex restrictions
  - Views as Items
  - Symmetrically associated attributes
  - Local Proxies

- Import & Export
    - Be RDF-compatible -- work and play nicely with other apps that use RDF
    - Be able to export:
        - export anything as RDF/XML
        - export calendar data as ical or xcal
        - export in other formats
    - Be able to import:
        - import some specific Chandler-formatted subset of RDF/XML
        - import calendar data as ical or xcal

For more background on associative data stores, see also:

- Mailing list messages:
    - MichaelLeQueux suggests an associative data structure like LazySoft
    - KevinSchroeder suggests an associative data structure like InterSys

---

- JonathanSmith wonders if anyone is paying attention to two other developments related to RDF issuess:
    - pytechnorati
    - Technorati API

Null Values

Some types of data stores support the idea of NULL values, while other types of data stores don't. For example, most relational databases support NULL.
Here's a quick summary how NULL values typically work in a relational database:

- by default, any record can have a field with a NULL value
- in the schema, a column can be required to be NOT NULL, in which case fields in that column cannot have a NULL value
- any field can have a NULL value, regardless of the required data type of the column
    - example: a text field can have a NULL value
    - example: a date field can have a NULL value
    - example: a foriegn key field can have a NULL value
- the NULL value is not itself an instance of any one data type, but it automatically matches any data type requirement
- a NULL value is always distinct from a "zero" value
    - example: in a text field, a NULL value is different from "" (a string of length zero)
    - example: in an integer field, a NULL value is different from 0
- new records are typically created with NULL values as the initial default values in many fields
- a NULL value in a field can be replaced with a normal value (example: NULL is deleted, and "Bob" is inserted)

- a normal value in a field can be replaced with a NULL value (example: "Bob" is deleted, and NULL is inserted)

Missing Attributes

In Chandler, different Items of the same Kind-of-Item tend to have the same attributes. For example, all the different Event Items will typically each have a "start time" and and "end time". Typically, each of the instances of an Event has attributes that match the Attribute Definitions in the attribute template defined for the Kind-of-Item called an Event. But that's not always the case...

Any given Event instance can have different attributes. In theory, I can create a new Event instance, and inspect it in some general purpose table view, and add a new attribute that only that particular Event will have. And similarly, I can delete an attribute from an particular Event, without deleting that attribute from all the other Event instances, or from the attribute template in the Kind-of-Item called an Event. So some Event instances can simply be missing some of the attributes that are nominally associated with an Event.
So that raises the question...

What's the difference between a missing attribute and an attribute with a NULL value?

What's the difference?

So if Chandler supports NULL values, and Chandler supports missing attributes, then do these two things have any semantic difference? For example, is there any real difference between these two Items:

Item A-1		Item A-2	
attribute	**value**	**attribute**	**value**
name	"purple jumsuit"	name	"purple jumsuit"
price	$40	price	$40
size	"medium"	size	"medium"
seller	"Sam"	seller	"Sam"
buyer	NULL		

One difference might be in how the user interface displays the Items. For example, in a general purpose table view, if you inspected Item A-1 alone, it would show up as a single row in a table with 5 columns. Whereas Item A-2 would show up as a single row in a table with 4 columns. In the table for Item A-1, the cell for "buyer" may be empty, but at least the column is there, inviting you to click in it and type something.

Will Chandler support NULL values?

I'm assuming Chandler will support NULL values. NULL values are a simple concept, but I think in practice they end up creating a lot of special cases in all sorts of code. The data store has to support some special representation for them, and serialization formats need to represent them, and query languages need to account for them, and

so on. But I'm assuming we're planning on having them, so this strawman data model makes frequent use of them.

...Proposed Strawman Solution...

- Have the data model support explicit NULL values.
- Have the data model support the specification of required attributes, which are attributes that are not allowed to have NULL values.
- New records are created with NULL values as the initial value in most fields.

Display Strings

@@@ -- Discuss this idea in the context of Global Attributes
Many Items have some kind of "display string". Display strings are usually short text strings, without tabs or line breaks. For example:

- an Email Message has a subject line, like "Weekly status report"
- an Event has a headline, like "Lunch with Tug"
- a Task has a title, like "mail 1040 form to IRS"

In the Calendar parcel, the python code can just be hard-coded to know which attribute has the headline, and so in any kind of summary list of Events, the user can see just the headlines. And similarly in the Email parcel. The Email code can hard-coded to know what attribute to use to display a list of messages.

But Items will also show up in general purpose views, like table views and outline views. Those views will have no idea what Kind-of-Item they're displaying, or what's inside the Items. So we maybe need some convention about display strings, so that if an Item wants to offer a display string, then a general purpose view can find out what the display string is.

This also comes up in the case of pop-up lists and selection dialogs. In some parts of the UI, the user may need to pick an instance from a list of instances. For example, a user may need to select a Group to send an e-mail message to, or select a Folder to add a Task to, or select a Priority Level to assign to a Bug. In some of those cases it would be nice to have general purpose pop-up lists and general purpose selection widgets, where the widgets are automatically capable of displaying any Kind-of-Item, without having to be configured to know what attribute to use for each Kind-of-Item. Here are a few options:

- have a single display name attribute
    - One option is to just have a standard attribute for the display name. Maybe call it or "title" (or "name", or "display name", or whatever). And then if an Item wants to be displayed properly in a general purpose view, it just needs to have a "title" attribute. So then all the Email Message items would have "title" attributes instead of "subject" attributes. And all the Contact items would have "title" attributes instead of "first name" and "last name" attributes.

    - But that sounds rigid. You could imagine that leading to problems when you start importing and exporting Items. When you Exported an Email Message to a foriegn e-mail format, the code would need to map the "title" attribute into a "subject" field, and vice-versa. That's maybe not a big deal in that scenario -- that e-mail export code needed to be hand-coded anyway. But what about when you're importing data in some more ad-hoc format. Let's say you're importing records from a comma-delimited text file, or from a relational database. In that case the attributes already have whatever names they have, and you don't want to go changing them. Which leads us to option two...

- specify a display attribute
    - A second option is to pick an existing attribute, and specify that that attribute should serve as the display string. So for an Email Message, the "subject" attribute might be the designated display attribute, and for an Event, the "headline" attribute might be the designated display attribute. How would the designation be represented in the data model? In the Kind-of-Item class, there could be a special attribute called "display attribute", and in each instance of Kind-of-Item the value of that attribute could be set to point to another attribute in the instance -- the attribute to be used as the display string.

- derived display strings
    - A third option is to have some kind of general purpose notion of derived attributes. For example, in an Event, the "duration" attribute might be a derived attribute, where "duration" equals "end time" minus "start time". And then for the purposes of display strings, a "display string" attribute could be derived from other attributes. So for a Contact, maybe ("display string" := "last name" + ", " + "first name"). And for Event, maybe ("display string" := "headline").
    - Of course, we can only have derived display strings if we have support for derived attributes. For more on that, see the next section, Derived Attributes.

...Proposed Strawman Solution...

- Have the data model support both the first & third options mentioned above.
- Have a special attribute named "display name", and have it defined in Item, so that all instances have it. Have it defined as holding an optional string literal. Instances of Item can have values for that attribute. Other Kind-of-Items can "refine" the "display name", and the refined versions can have derivation rules for deriving the "display name" from other attributes.

Derived Attributes and Derived Items

I don't know whether there's been any talk of Chandler having support for derived attributes. Derived attributes might well be beyond the scope of the project. I have a feeling they're just the "tip of an iceburg", and that they would lead to great deal of complexity. But, if we do want to have derived attributes, then they need to be represented in the data model.

Just for context, here's a quick repeat of the example that I gave in the section above, in case you didn't read that:

- *For example, in an Event, the "duration" attribute might be a derived attribute, where "duration" equals "end time" minus "start time". And then for the purposes of display strings, a "display string" attribute could be derived from other attributes. So for a Contact, maybe ("display string" := "last name" + ", " + "first name"). And for Event, maybe ("display string" := "headline").*

There are a lot of questions about how derived attributes would be implemented. Here are a few of the questions:

- Should derivation rules be associated with instances or classes?
  - One way to handle derived attributes would be to associate them with individual instances. For example, if we wanted Events to have display strings, then each and every Event would have its own "display string" attribute, each with identical derivation rules as values. Any given Event could have some different derivation rule, although in practice that wouldn't come up much.
  - Alternatively, the derivation rule could be an attribute of the Event Item itself (the Kind-of-Item called "Event", rather than the instances of "Event"). That way the derivation rule would only be stored in one place, which (a) would save space, and (b) allow you to edit the rule in just one place but have it take effect "globally". But to implement that, we'd need some kind of new mechanism for finding the attributes of an Item. Right now all the attributes of an Item are associated directly with the Item. But in this new scenario, some attributes would be associated with the Item directly, while others would be associated with the Kind-of-Item that the original Item is an "instance of". So, for example, looking up the value of an attribute would require not just a dictionary look-up, but also a little walk up the "instance of" relationship and a look-up in another dictionary. And the same thing goes for getting a list of all the attributes of an Item.
  - One further alternative would be to do **both** of the above alternatives. A derivation rule could be associated with an Event Item itself (so that the derivation rule is "inherited" by all the instances of Event), **and** individual instances of Event could also have their own associated derivation rules, which would override any identically named "inherited" one.

- When should derivation rules be evaluated?
  - Maybe whenever something asks for the value of the attribute?
  - Maybe whenever there's a change in the value of one the attributes that's mentioned in the derivation rule?
- Is the result of the evalution cached?
  - Cached in a python object?
  - Cached in a repository?
- How do we resolve circular references between two (or more) derivation rules?
- How does the UI indicate that an attribute is derived?
- What UI does a user use in order to create a derivation rule?

- If Chandler has a generic table view, and the cells can have derivation rules, then is Chandler a spreadsheet? (And is that maybe biting off more than OSAF can chew? ;-)

Derived Items

- @@@ -- Discuss the issue of derived items. Not just objects with attributes that are derived, but entire objects that are entirely derived -- e.g. birthdays & infinitely recurring events

...Proposed Strawman Solution...

- @@@ -- need to identify how this impacts other parts of the overall architecture

- Have the data model support derivation rules.
- Derived results can be cached in the repository.
- A derivation rule can be associated with an Attribute Definition.
    - Example: The Kind-of-Item named "Event" can specify that all Events have a "duration" with the derivation rule: "duration" := "end time" minus the "start time".
- A derivation rule can **not** be associated with an individual Attribute Value.
    - Example: The Event "Go to Safeway" can **not** have a "start time" with the derivation rule: "start time" := "end time" of the "Do laundry" Event.

Default Attribute Values

I don't know whether there's been any talk of Chandler having support for default values for attributes. If we want to have default values, I can think of a couple different ways that they might work:

- initialization values
    - When a new Item is created, Chandler could initialize the attribute values to default values. For each new attribute, the initial value could be a copy of the default values specified in the corresponding attribute in the Kind-of-Item that this new Item is an "instance of".

- overriding values
    - When a new Item is created, Chandler could initialize the attribute values to some special value like UNSET. The UNSET value would be a special value, similar to how NULL is a special value. Like NULL, the UNSET value would automatically match any data type restrictions specified for an attribute. When Python code asks for the value of an attribute, the value UNSET will never be returned. Instead, if the value is UNSET, then the Item checks to see what the default value is for the corresponding Attribute Definition in the Kind-of-Item that this Item is an "instance of", and returns that default value.

If Chandler does end up supporting both Default Attribute Values and Derived Attributes, then I think the implementations will end up being related. Either both implementations will copy attribute info into new Items whenever a "create" method is called, or both implementations will walk up the "instance of" relationship to look up values whenever a "get" method is called. In either case, there's likely to be just one set of code that does the work.

...Proposed Strawman Solution...

- Have the data model support default values by using the "initialization values" technique described above.
- Do not support the idea of UNSET values

Sub-Attributes

- KatieCappsParlante, 24 March 2003
  - RDF has the concept of one property being a subproperty of another, which we're interested in. An example: "father", "sister", "aunt" properties are all sub-properties of "family", which is a sub-property of "relationship". The "Katie" instance has a "brother" attribute, which has a value of "Rob". She also has a "husband" attribute, which has a value of "Nick". If we do a query for all "family" attributes of "Katie", we get both "Nick" and "Rob". In RDF-land, they usually use examples like "Creator", "Author", "Director", "Painter", "Actor", etc.

- BrianDouglasSkinner, 24 March 2003
  - Do subproperties inherit the restrictions of their parent properties? (I'm guessing yes to this one.)
  - Do subproperties inherit default values? (I'm guessing no to this one.)
  - Can a super-property ever be an "abstract" super-property? Meaning, do you ever restrict people from a assigning a value directly to the super-property?

- KatieCappsParlante, 25 March 2003
  - Subproperties inherit restrictions: not necessarily.
  - Subproperties inherit default values: unlikely
  - Abstract superproperty: unlikely, I think this adds complexity without much value
  - Mostly, I see super/sub properties as being useful for queries, perhaps polymorphic uis. (e.g. some view shows a list of all "family" members for a given "person"). Its not really the query that's using the sub/superproperty concept, but the view.
  - In spirit, we probably don't want to go overboard with the "restrictions".

- KatieCappsParlante, 25 March 2003
  - btw, when we say that an attribute is hierarchical, we mean that the schema definition of the attribute is hierarchical. When we say

attributes are global, we mean they are defined globally. Attribute values are still tied to an instance.

- see also the example below

- See also DB Topics: Property Types

...Proposed Strawman Solution...

- Have the data model support sub-attributes.
- Sub-attriubtes do not inherit restrictions.
- Sub-attriubtes do not inherit default values.
- Attributes can never be abstract.

Global Attributes

- KatieCappsParlante, 24 March 2003
  - Usually in OOP, attributes are defined in relation to a class ...[snip]... We definitely want to define attributes globally in chandler, with the idea that these attributes are building blocks that people can use to extend their schema, and that the attributes will be useful for interesting queries.

- BrianDouglasSkinner, 24 March 2003
  - In practice, how much will global attributes serve as templates, versus how much they will be used in ways that rely on their having unique, shared identities?
  - Are there any differences, or subtle distinctions, between having an attribute be a global attribute, versus having the attribute be an attribute of Item? In both cases, the attribute is defined in the schema, and a user can make a new Item that has a value for that attribute.
    - Here's one potential difference. If an attribute is an attribute of Item, then a generic data entry form, like a generic table view, is likely to offer to display the attribute, along with all the other attributes of an item. Whereas with a global attribute, a generic data entry form will probably not prompt the user to assign a value for that attribute, unless the user says that this particular item has that attribute.

- KatieCappsParlante, 25 March 2003
  - btw, when we say that an attribute is hierarchical, we mean that the schema definition of the attribute is hierarchical. When we say attributes are global, we mean they are defined globally. Attribute values are still tied to an instance.
  - Example:
    - onDate is an item
    - onDate is an attribute
    - onDate attributes always have datetime values

        -
        - startTime is an item
        - startTime is an attribute
        - startTime is a subattribute of onDate
        - startTime attributes always have datetime values
        -
        - event is an item
        - event is a class
        - event instances have startTime attributes
        -
        - note is an item
        - note is a class
        -
        - foo is an item
        - foo is an instance of the event class
        - foo has a startTime of 12/12/2003 11 AM
        -
        - bar is an item
        - bar is an instance of the note class
        - bar has a startTime of 1/3/2003 12 PM
        -
        - blah is an item
        - blah has a startTime of 3/4/2003 1 PM
        -
        - So, even though we have said nothing about notes having startTimes in our "schema", the bar instance still has a startTime. Going even further, blah is just an item, with no class providing schema information. Blah can still have a startTime. We've made schema assertions about the startTime attribute, but no schema assertions about blah. In this case, everything we say about events, notes, and startTimes applies globally.
    - Alternatively, the "restrictions" could apply just to the class, not globally. (Although, in this case, it seems like you actually want the restriction to apply globally):
        - onDate is an item
        - onDate is an attribute
        -
        - startTime is an item
        - startTime is an attribute
        - startTime is a subattribute of onDate
        -
        - event is an item
        - event is a class
        - event classes have startTime attributes
        - **when event classes have startTime attributes, the value is a datetime**
        -
        - note is an item
        - note is a class
        -
        - foo is an item
        - foo is an instance of the event class

- foo has a startTime of 12/12/2003 11 AM
-
- bar is an item
- bar is an instance of the note class
- bar has a startTime of 1/3/2003 12 PM
-
- blah is an item
- blah has a startTime of 3/4/2003 1 PM

...Proposed Strawman Solution...

- @@@ -- We need to identify how this impacts other parts of the overall architecture.

- Have the data model support both global attributes and attributes that are defined "within" the context of a Kind-of-Item.
- Have the data model support non-global attributes. Non-global attributes are defined "within" a single Kind-of-Item, and they are "owned" by that Kind-of-Item.
- Have the data model support Item-centric attributes. Item-centric attributes are defined "within" a single Item, and they are "owned" by that Item.
- Have the data model support global attributes. Global attributes can be defined in a Parcel, and can be used by any Kind-of-Item or any instance of an Item. Global attributes can be defined in a Parcel, but they are available "outside" the Parcel, to any Kind-of-Item being defined anywhere else.

Enumerated Types

@@@ -- Figure out how to add support for hierarchical ENUMS

An enumerated type (also called an ENUM) is a simple data type, where the definition of the data type includes a complete list of the possible values that can be assigned to an attribute of that type. For example, you could create an ENUM called "Day of the Week", where the list of all possible values was defined to be (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday).
Here's a quick summary of how ENUMs are handled in most systems:

- The initial list of values is defined along with the datatype.
- It is impossible to create new "instances" of the type after the type is defined.
- Each instance typically has name -- e.g. "Monday".
- It's usually impossible to define any attributes other than the name.
- The instances typically have an order. It's an ordered list of values, not just a set of values. For example, the weekdays are (Sun, Mon, Tue, Wed, Thu, Fri, Sat), NOT (Thu, Mon, Wed, Tue, Sun, Fri, Sat).

ENUMs in forms

ENUMs have lots of good uses. One of the places ENUMs work nicely is with forms. Once an ENUM has been defined, the system can automatically create pop-up lists

showing the possible values. For example, in a bug-tracking database, the bug-entry form might have a field for "priority", with a pop-up list showing the choices ("High", "Medium", "Low").

In a Chandler generic table view, a cell with an ENUM value could have widget that brings up a pop-list when the user clicks in that cell. For example, in the Quicken application, you can see a table of transations, and Quicken has a nice UI widget for selecting the "Category" of the transaction.

Requirements

- Chandler needs to have support for ENUMs
- Chandler needs to have support for hierarchical ENUMs.
- It must be possible to assign a defined order to a set of ENUM values.
- We may want to have more than one possible defined order for a set of ENUM values. Maybe we want to have named orderings.
- ENUMs do **not** have a fixed set of instances. With the right permissions, a user can add more instances to an ENUM.
- ENUMs do **not** have a fixed set of attributes. With the right permissions, a user can add more attributes to an ENUM.

Implementation Options

Here are two different options for how the data model might represent ENUMs:

- **support for explicit ENUMs**
  - The Chandler data model could specifically model ENUMs, and Chandler could have special code to handle ENUMs.
  - For example, a user could create a "Bug Priority" ENUM with values ("High", "Medium", "Low"). And then in the Kind-of-Item called "Bug", the user could create an attribute called "priority", and restrict the "priority" attribute to only having values of type "Bug Priority". In any part of the UI, whenever a user is editing a "Bug", if the user wants to change the "priority" then Chandler automatically offers a pop-up list of "Bug Priority" values. Optionally, Chandler could automatically restrict the user from ever assigning anything other than a "Bug Priority" value to the "priority" attribute of a "Bug".
  - Chandler could also have support for hierarchical ENUMs. Quicken, for example, offers hierarchical ENUMs for the "Category" of a transaction.

- **support for using a Kind-of-Item as an ENUM**
  - Rather than having the Chandler data model specifically model ENUMs, we could instead set things up so that any Kind-of-Item could serve as an ENUM.
  - ENUMs are really just a sort of "lightweight" type of class. ENUMs may have a small set of instances and a limited number of attributes, but in most respects they behave just like an ordinary Kind-of-Item:
    - users can create types of them
    - users can create instances of them
    - parcels can create them
    - other Items can have instances of them as attribute values

        - attributes can have types of them as the required type for the attribute
        - schema discovery requests can return types of them
        - queries can return all the instances of them for a given type of them
        - etc.
    - For example, a user could create a Kind-of-Item called "Bug Priority", with an attribute "name". And then the user could create three instances of "Bug Priority", with the names ("High", "Medium", "Low"). And then in the Kind-of-Item called "Bug", the user could create an attribute called "priority", and restrict the "priority" attribute to only having values of type "Bug Priority". Now we just need a couple more features, in order to get our "Bug Priority" set up to behave exactly like the "explicit ENUM" described in bullet point 2 above.
        - For one thing, when a user is editing a "Bug", if the user wants to change the "priority" then we need Chandler to automatically offer the pop-up list of "Bug Priority" instances. Chandler needs to somehow know that there's a small, fixed set of instances. One way to do that is in the instance of Kind-of-Item called "Bug Priority", just have an attribute that points to all the instances. Actually, in principle it would be nice if every Kind-of-Item had a list of all its instances, or at least pretended to have a list of all its instances, even in that list had to be cons'ed up by the repository in response to a request. But you could imagine a solution where each Kind-of-Item at least starts out with a list of all its instances, and only abandons the project if the list gets to large to manage. And then in the UI, whenever a user goes to change the value of an attribute, if (a) the attribute is restricted to be some other Kind-of-Item, and (b) there are some modest number of instances of the target Kind-of-Item, then (c) the UI offers a pop-up list of the instances. (Actually, even if (b) isn't true, the UI should have some affordance for helping the user select one of the instances.)
            - Note: If a Kind-of-Item keeps a list of its instances, that has implications for how garbage collection works. See Garbage Collection.
        - That brings us to the problem of ordering the instances. In the pop-up list, the instances should show up in order. A good list looks like ("High", "Medium", "Low"), not ("Medium", "High", "Low"). So in the data model, the Kind-of-Item needs to somehow know the order of its instances. How that's actually done is "an implementation detail" for later. For now, the question is how the data model APIs should expose this. Some Kind-of-Items need to have ordered lists of instances, while others can just have sets of instances, and others still maybe won't ever even need to return any kind of collection of all their instances. API-wise, "Ordered vs. unordered" could maybe be a property of the "instances" attribute on the Kind-of-Item. Or, alternatively, Kind-of-Item could have a separate attribute for specifying that the instances are ordered (e.g. a boolean attribute named "instances are ordered"). And then there's another API issues: if a Kind-of-Item has ordered instances,

then when a new instance is created, the process of creating it must also specify where it should appear in the ordering.
    - Optionally, if we really want to get a Kind-of-Item to behave exactly like an ENUM, then we might want to be able to put a couple restrictions on the Kind-of-Item. We might want to make it be impossible to add new attributes, or to create new instances. Each Kind-of-Item could have flags for those restrictions. For example, the existing "abstract" flag can be used to mark a Kind-of-Item as an abstract superclass, so that the repository does not allow it to be instantiated. There could be a similar "no new instances" flag, that would restrict the system from allowing new instances to be added to an ENUM. And there could be a "no new attributes" flag as well. Or simply a single "ENUM" flag, which would mean both "no new instances" and "no new attributes".

Right now my vote is for the third option. The first option (no support) seems out of the question. Between the second and third options, I think the second will end up being more complicated and clunky. The third definitely creates some complexities, but I think in the long run it will work out better. That's my two cents. The rest of this strawman data model basically assumes the third option, although the data-structures here gloss-over some of the important details.

Discussion

- Chandler will definitely support enumerated types, and it will need to support hierarchical enums (e.g. the quicken example you mentioned). In the short term we were thinking of trying to get away with enums being strings (or a list of strings or list of strings, to get at the hierarchical feature), but we've also talked about what you're calling "Kind-of-Item as ENUM". I was going to make an argument that your Enum-as-class solution would be more cumbersome than an Enum-as-type solution, but thinking on it more carefully I think they're perhaps nearly the same thing. The ENUM type itelf should be stored as an item, lets put it that way. -- KatieCappsParlante, 24 March 2003

...Proposed Strawman Solution...

- Have the data model support ENUMs, using the technique described above under the heading "support for using a Kind-of-Item as an ENUM".

Unique IDs

@@@ -- this section still needs to be written up
@@@ -- Discuss the data model impact of the issue about ids as single (UUID) vs. tuples (repository ID, local ID)

- UUID discussion from the dev mailing list:
    - http://lists.osafoundation.org/pipermail/dev/2003-April/000535.html
    - http://lists.osafoundation.org/pipermail/dev/2003-April/000538.html

Item Versions

@@@ -- this section still needs to be written up
@@@ -- Discuss the data model representation of item versions. Lots of options and issues:

- Save list of item versions?
- Save list of (forward) change objects?
- Save list of backward change objects?
- Save version number in the current item?
    - Add a _p_serial attribute to Item? See the last paragraph of http://lists.osafoundation.org/pipermail/dev/2003-April/000549.html
- Discuss use of tumblers to track version histories, and recognize uid collisions
    - http://lists.osafoundation.org/pipermail/dev/2003-April/000539.html

---

Namespaces

The idea of namespaces shows up in lots of different kinds of computer system. RDF has an idea of namespaces. Java has an idea of namespaces. And, in a way, even the Windows file system has namespaces.

In the Windows file system, files live in directories. A file in a directory is uniquely identified by its file name (e.g. "Report.doc"). Within a directory, no two files can have the exact same name. But two files in different directories can both be named "Report.doc". Each directory serves as a namespace for the file names in it. If you know that the file is "C:\August\Report.doc", then you know the name of the namespace and the name of the file, and you have enough information to find the file and distinguish it from all the other "Report.doc" files on the hard drive.
RDF namespaces work pretty much the same way. In the context of local namespace, an RDF attribute can have some simple name, like "width". But there might be a lot of other "width" attributes out there, created by different people, with different meanings in mind. If you refer to the attribute using both the namespace and attribute name, then you can uniquely identify the specific "width" attribute that you want to use.

Does Chandler need namespaces?

Chandler users will create a lot of new Kind-of-Items. A user in a biology lab may create a new Kind-of-Item called "Virus", to keep track of HIV strains. A user at a software company might create a different new Kind-of-Item called "Virus", to keep track of computer viruses. But the two "Virus" definitions are in two separate repositories, so everything is fine. But then over time, the two definitions may come to live in the same repository. The software company finally ships its anti-virus Chandler parcel, and the user in the biology lab loads the parcel into their Chandler repository.

Is that a problem, having one repository with two different Kind-of-Items, both with the same name? No, it's not a problem as far as Chandler is concerned. Each "Virus" has its own uid, so Chandler knows how to tell them apart. Other instances in the repository may have a reference to one of the Viruses, but the reference is always stored as a uid, not as a text string, so the other instances will never be confused about which Virus is which. It might be confusing for users, but it won't be a problem for Chandler.

Do users need namespaces?

We want to keep users from being confused by instances with identical names. We might want to artificially impose some of the idea namespaces, just for the sake of the user. For example, we could impose a requirement that within each Parcel all the Kind-of-Items have to have unique names. And then in the UI the display that shows the Kind-of-Items could be sorted by Parcel, so that the user can see that one "Virus" is a "Biology Lab" type, and the other "Virus" is an "Acme Fix-it" type.

Namespaces for Attributes?

Just like with Kind-of-Items, Chandler shouldn't need namespaces for attributes either. At least, strictly speaking, Chandler shouldn't need namespaces for attributes. In theory, all the Chandler code should be able to reference attributes by uid rather than by name. But in practice it may be useful to index into attribute dictionaries by name. And, users would probably find it confusing if an Item could have two different attributes, both with the same name. So Chandler should probably impose the requirement that attribute names need to be unique within an Item (and therefore also unique all the way up and down the inheritance chain for this Item).

String-based References

For the question of whether or not Chandler need namespaces, the answer depends on whether or not Chandler ever uses string-based references. If attribute dictionaries are indexed by attribute name, with a string, then namesapces become more important.

My thinking right now is that we should try to set up the Chandler repository to **never** use string based references. I think they lead to a host a problems. For example, string-based references create new issues for garbage collection. For more about that, see the discussion about string-based references in the section below about Garbage Collection.

My recommendation is to never use lexical names to look up instances. Instead, all references should be handled using UIDs, in the repository, and in the Chandler code and data structures.
Just for fun, here's an amusing example of a lexical-reference problem. Back around 1990, NeXT Computer created an Objective-C class called NSButton. Objective-C didn't have namespaces, so NeXT needed some way to differentiate their Button class from all the other potential Button classes. NeXT created a psuedo-namespace for themselves by simply prefixing all their classes with "NS", so that "Button" became "NSButton". Eventually NeXT got bought by Apple, and Apple took the old NeXT code and turned it into Cocoa. So now, in 2003, Apple is shipping a Cocoa class called "NSButton". The "NS" doesn't stand for "Apple" or for "Cocoa", and now it

doesn't even stand for "NeXTSTEP" anymore. It's just cryptic, archaic cruft. But there's no way to change it now, because the string "NSButton" is hard-coded in thousands of files, across hundreds of companies. Oops.

Discussion

- I was assuming the use of namespaces. I'm not convinced that uids do away with even chandler's need for namespaces. Will the "event" kind-of-item have the same uid in everyone's repository? I'll have to think about this one more carefully, but interesting points to bring up. I was assuming there'd be a "chandler" namespace for the core schema and some core objects -- whatever the app starts out with -- and then a user namespace for new items. There's the problem of a unique namespace for the user, but I think that could be solved with email address, jabber id, or some other solution. I'd think it needs to get solved for other reasons anyway. -- KatieCappsParlante, 24 March 2003

- "Will the "event" kind-of-item have the same uid in everyone's repository?" --> see also DbOidCoinageProblem

- See also the page NameSpaceIssue, which raises the questions:
  - *For user defined items, how do we prevent name space collisions? Do we need a name space hierarchy (e.g. anEmail.osaf.project=chandler)?*
  - *Are all user defined attributes, attributes of Item (and not for example, Event or Contact)? Can a contact have a user defined attribute that is not defined for an event? If so, can the contact and event have the same attribute 'name' but different semantics?*

...Proposed Strawman Solution...

- The Chandler code and data structures never use lexical names to look up instances. Instead, all references are handled using UIDs.
- Chandler enforces some namespace restrictions, solely for the sake of the user.
  - Within a Parcel, Kind-of-Items must have unique names.
  - Within a Parcel, global attributes must have unique names.
  - Within an Item, attributes must have unique names.

Semantic Dictionaries

@@@ -- this section still needs to be written up

The idea behind semantic dictionaries is to have some data structure in the Chandler repository that would provide a mapping between Chandler data and third-party semantics. For example, the iCalendar standard would be one third-party semantic we care about.

See also: SharedSchemaIssue

Restrictions on Attributes

One of the reasons for having a schema is to impose restrictions on a attribute, to limit what kinds of values can be assigned to that attribute.
For example, in an Event, the "start-date" should always be a date. The user should **not** be allowed to create a new Event and set the "start-date" to "yogurt". If the "start-date" did get set to "yogurt", then the Event would never match any query generated by Day View, or Week View, or Month View, or any other normal calendar view. And so the Event would drop out of sight of user, lost to them unless they have it bookmarked, or make some generic table view query that returns it.

You could argue that this is just the sort of flexibility that Chandler is supposed to have, but I think Chandler should also have the "flexibility" to let some attributes be strongly typed. Maybe the user should sometimes be able to override a restriction, but Chandler should still know about the restriction and at least pop up panel to warn the user and ask them if they really want to do that.

See also DB Topics: Schema Validation.

Types of restrictions

The most common type of restriction is one that restricts the data type that can be assigned to an attribute. For example, a "start-date" has to be a date, or "annual rainfall" has to be a float, or "father" has to be a Person.
Cardinality is another common type of restriction. For example, a Department can have many "students", but an Employee can have only one "salary". The most common cardinality restrictions are "1" and "Many", but there are other possible cardinality restrictions. In relational databases, a field can be restricted to be "NOT NULL", which is a way of saying that the cardinality is required to be "1", rather than "0 or 1".

But there can be other types of restrictions too. For example, "end date" could have the restriction that it must fall after "start date", and "annual rainfall" could have the restriction that it must be non-negative, and "father" could have the restriction that a Person's "father" cannot point to any Person's "children".

Where are restrictions enforced?

Okay, so in Chandler, what code is going to enforce the restrictions? And where will the restrictions be expressed?

One obvious option is to do this all in Python code, on the client. An Event class serves as the encapsulation of all the knowledge about how Events should behave. The Event class should know that the "end-date" value must fall after the "start-date" value. That's probably the traditional object-oriented thing to do.

That works well in traditional object-oriented apps, but unfortunately, in Chandler it may not work. Part of the problem is that Chandler will have general-purpose views, the generic table view. So a user may end up editing a "start-time" in a generic table view, rather than in a calendar view like Week View. That would be still be okay, as

long as the generic table view was setting the "start-time" via the Event object. But we can't guarentee that.

In Chandler, the calendar data may be shared. Alice may have installed the Calander parcel on her machine, and created an Event with a "start-time". Alice publishes the Event, and then Bob views it in a generic table view on his machine. Bob has never installed the Calander parcel on his machine, and his Chandler client doesn't even know that there is a custom Python class that represents Events. So when Bob edits the "start-time", his edit is handled by a general-purpose Item class.

Here are a few different options for enforcing restrictions on "start-time":

1. We could refuse to let Bob edit the "start-time", because he's trying to make the edit from outside the Calendar parcel, and we know that Events are **owned** by the Calendar parcel, so only Calendar parcel code is allowed to edit them. That solution probably isn't in the spirit of Chandler.

1. We could let Bob edit the "start-time" however he damn well pleases. If he wants to change it to "yogurt", then more power to him. And then in the Event class, and in the rest of the Python code in the Calendar parcel, we can have a great deal of error-checking code. All the code would be carefully written to avoid making any kinds of assumptions about restrictions on attribute values. The code would gracefully handle exceptional conditions, and display Events to the best of its ability.

1. We could vigorously enforce restrictions on "start-time", but enforce those restrictions outside of the Calendar parcel Event code. The restrictions could be enforced on the client side, in a general-purpose Item class. Or the restrictions could be enforced by the repository, and editing changes submitted to RAP could be refused, with an error code that cites the restriction that was violated.

Restrictions enforced in the repository

If Chandler does end up having restrictions be enforced by the repository, then the restrictions necessarily have to be expressed in the repository, rather than be expressed in Python code in classes like the Event class. And actually, even that's true even in the case where the restriction is enforced on the client, but enforced by some general-purpose Item class. For any kind of general-purpose code that enforces restrictions, the code needs to be able to load a set of restrictions, and those restrictions should be stored in the repository. And if you're going to do that, then maybe you might as well have the restrictions actually be enforced in the repository. That way the repository isn't relying on the client to have been coded correctly. After all, the client may not be a dedicated Chandler client; it might be some hastily coded conversion routine made for shovelling data from some legacy system over into a Chandler repository.

Issues

So, after all that discussion, that still leaves us with a lot of unresolved issues:

- What types of restrictions should Chandler support?

    - Data type restrictions?
    - Cardinality restrictions?
        - One vs. Many cardinality restrictions?
        - NOT NULL restrictions? (same thing as required vs. optional attributes?)
        - Other cardinality restrictions? (e.g. min and max?)
    - Range restrictions?
        - example: "rainfall" must be a **non-negative** number
        - example: "age" must be over 21
    - Inter-Attribute restrictions?
        - example: "start-time" must come before "end-time"
    - Inter-Item restrictions?
        - example: my "mother" must have a "birth-date" prior to my "birth-date"
    - Transitive restrictions?
        - example: my "mother" cannot be the "daughter" of my "son"

- How should restrictions be represented?
    - Hard-coded in Python code?
    - In the schema data in the repository?
    - Other?

- Where are the restrictions enforced?
    - Nowhere?
    - On the client?
    - In the repository?
    - On the client **and** in the repository?

Discussion

- So we have this question of where restrictions should be enforced. That question overlaps a little with a larger architecural question about what parts of the system should be dumb and what should be smart. For some discussion of that, see DbEndToEndDumbData.

- You describe restrictions on attributes (cardinality, type), presumably globally. That is, if a starttime attribute has a value it is always a datetime, wherever it shows up. Usually in OOP, attributes are defined in relation to a class, and any restrictions apply to that attribute-class pair. We definitely want to define attributes globally in chandler, with the idea that these attributes are building blocks that people can use to extend their schema, and that the attributes will be useful for interesting queries. One could imagine defining the **restriction** in terms of the attribute-class relationship, though. The starttime attribute has a cardinality of 1 when applied to an event, but might have a cardinality of many when applied to some other class. I'm kind of on the fence on this issue. The RDF folks seemed to think the global restrictions could cause problems, but its hard for me to find a compelling example in our pim space, especially for the most interesting restriction, type. -- KatieCappsParlante, 24 March 2003

- see also the example above

- We support data type restrictions for attribute values, and some Attribute Definitions can specify that the value must be a text literal. For any Attribute Definition that specifies a text literal value, should we actually allow that Attribute Value to point to any type of Item whatsoever?
    - pro: Sure, why not? What could that possibly break?
    - con: Are you crazy? Then what was the point of even specifying the restriction in the first place?

...Proposed Strawman Solution...

- @@@ -- We need to identify how this impacts other parts of the overall architecture.
- @@@ -- Maybe restrictions can be handled as "hints". Hints could be stored in the repository, but the repository wouldn't need to actually enforce the hints.

- Represent restrictions within schema data the data model.
    - Represent simple data type restrictions on the Attribute Definition.
    - Represent simple cardinality restrictions (One and Many) on the Attribute Definition.
    - Represent simple required (NOT NULL) restrictions on the Attribute Definition.
    - For all the other more complicated restrictions, have the Attribute Definition item refer to an Attribute Restriction item. For now, punt on actually trying to figure out the representation of complex restrictions within an Attribute Restriction item.

Symmetrically Associated Attributes

For an introduction to symmetrically associated attributes, see Steve Zagieboylo's design list post. I'm also a fan of symmetrically associated attributes. I included them as a feature in a database product that I wrote back years ago. They were simple and they worked great.

In the strawman data model outlined below, I've included support for symmetrically associated attributes as an optional feature. The data model allows for having some attributes which are specified as having symmetrically associated attributes and other attributes which are traditional uni-directional references. Of course, this is the easy part. It's easy to write a data model design doc that supports symmetrically associated attributes. It's another thing all together to incorporate that feature into the repository!

Here's the relevant excerpt from Steve's post:

There was a special type of attribute we called an association, in which items of one class were related to items of another class. This was like the links that your RDF data store has, except that they were always kept balanced by the database. For example, you might have a class "Person," and another class "Project." The Person class might have an attribute "WorksOn" and the Project class could have an associated attribute "Workers." The data in association attributes were always

symmetrical: For example, if I added the Project ABC to Fred's WorksOn list, then Fred was automatically added to ABC's Workers list. This symmetry was built into data store.

The class "Class" also had an attribute "Attributes" which contained the list of attributes (that is, the instances of class Attribute) for that class. Since the class "Class" was also an instance of class "Class," it had a value for the attribute "Attributes:" a list that included "Parent, Children, Instances, Attributes," and more. The Attribute class (that is, the instance of the class "Class" which describes all Items of the class "Attribute") also had, of course, a list of attributes. These included "ForClass, Type, and Associate."

The value for "Associate" was critical for the system to maintain the association symmetry. For example, the attribute "Person.WorksOn" was an associate of the attribute "Project.Workers." This is how, when Project ABC was added to Fred's WorksOn list, the system knew to add Fred to Project ABC's Workers list. Associations could be many-to-many, many-to-one, one-to-many, and one-to-one. (Note how a single change to a one-to-many value might change values in a total of ***three*** *items. i.e. If Fred currently ReportsTo Mary and I add Fred to Sue's DirectReports list, then the Items for Sue, Fred, and Mary all change.)*

Many of the attributes in the fundamental Items had associates. {Class.Instances ... Item.Class}, {Class.Attributes ... Attribute.ForClass}, and {Class.Children ... Class.Parent} were all one-to-many ... many-to-one pairs. Now for the quiz: The attribute "Associate" was, of course, an instance of the class "Attribute" and represented an association. All associated attributes have an associate, and this one is no exception. What was the associate of the attribute "Associate?"
[Answer: "Associate"]

Discussion

- It's possible that the idea of symmetrically associated attributes may having some bearing on distinguishing between strong references and weak references. See Garbage Collection.

- @@@ -- Discuss the issue of representing many-to-many relationships

- @@@ -- Discuss the issue of representing cross-repository symettrically associated relationships

...Proposed Strawman Solution...

- Have the data model include optional support for symmetrically associated attributes.
- Don't require all inter-item relationships to have symmetrically associated attributes.

Garbage Collection and Deletion

Here's a quick summary of garbage collection. It's an excerpt from DB Topics: Garbage Collection:

- *A database should support garbage collection, which means content no longer reachable from an application's data set should be reclaimed by the database, without explicit intervention from the application. This frees application programmers from worrying about manual memory management. In practice, this means a database must understand inter-object references, so that the transitive closure of references from a root will keep content alive. Garbage collection is the process of reclaiming unused space in a storage system, by scavenging any space that is not reachable from a set of garbage collection roots. Typically a storage system has well-known entry points where content is mounted, and these entry points are the roots for deciding what content is regarded as inside the system. The content in a database that can be reached by following references from a garbage collection root is considered alive, and will not be scavenged during the next garbage collection. Whereas content unreachable from the roots is considered garbage and can be omitted from the database at the discretion of the system reclaiming space.*

And here's some discussion about strong an weak references. It's an excerpt from DB Topics: Reference:

- *Python object pointers tend to be strong references, which keep preferred objects alive. In contrast, URL based references are typically weak references, which might become stale and 404 would be referenced. An assumption about which of these occurs pervasively in a database will dramatically impact the style of an application following references from one object to another. On a related topic, string based references can be made more efficient in many contacts by using interned strings (OsafDbInterningStrings?), which reduces space overhead and reduces time for comparison. The use of URLs as the identity of objects is described as a separate topic DB Topics: URL Identity.*

See also the related topics:

- DB Topics: Delete vs. Kill.
- Item Versioning.

Issues

Okay, so if we have garbage collection in the data store, that might have some impact on the data model.

- Should the data model distinguish between strong and weak references? Should the schema allow for both types?
- Will string-based references be used in some places? Will these be treated as weak references? Will the user be allowed to change the name of an item if that name is the target of a string-based reference? Will changing the name break the references, or will all the references be updated accordingly? In addition to string-based references within a repository, we need to think about string-based references stored completely outside Chandler, in word processing documents and browser bookmark files. See also DB Topic: URL Identity. And note the connection between this issues and Namespaces.

- Could symmetrically associated attributes represent strong references, while unilateral attributes represent weak references?
- What does it mean to have garbage collection in a system that keep track of version histories? If old versions of items reference old versions of items, how do items ever become un-referenced?
- @@@ -- Discuss the issue of deletion semantics and relationships to wholly-owned items

...Proposed Strawman Solution...

- Never use string-based references. Just say no.
- Items have a "deleted" attribute, that can be used to mark that they have been logically deleted.
- (@@@ -- Should the schema allow for both strong and weak references?)

Local Proxies

@@@ -- this section needs to be re-drafted to take into account the idea of relay servers

Repositories can contain proxy-copies for items that come from other repositories.

For example, while a user is working offline, a local repository in a laptop may have proxy-copies for most of the items that are canonically stored in a server repository. Or, when one user shares a calendar with another, an item in the repository of the shared calendar may be proxy-copied into the repository of the subscribing calendar.

Proxy-copies vs. copies

A proxy-copy is different from a regular copy. In Microsoft Word, when you copy and paste a paragraph from one document to another, the pasted paragraph doesn't know that it's a copy, and it doesn't keep any kind of pointer back to the orginal paragraph in the original document. In Chandler you can also copy and paste paragraphs. And you can copy and paste Items. Just like in Microsoft Word, when you copy and paste an Item in Chandler, the pasted Item doesn't know that it's a copy, and it doesn't keep any kind of pointer back to the orginal Item. The new pasted Item will have its own unique uid, completely unrelated to the uid of the orginal Item.

In contrast, when a proxy-copy is made, the new proxy Item is created with a pointer that points back to the original Item. Chandler can use that pointer to keep the two Items in sync. Changes to the proxy can be folded back into the original, and changes in the original can be applied to the proxy.

Pointers to original Items

Each proxy Item is created with a pointer that points back to the original Item. That pointer could be represented in different ways, and I'm not sure what's the best way to do it. One alternative is for the proxy Item to have a special attribute like "copied

item", that points to the copied Item. If Item-A has a "copied item" attribute with a reference to Item-B, then we now that Item-A is a proxy-copy, and we know that it's a copy of Item-B. If Item-C has a "copied item" attribute with no value (i.e. a NULL value), then we know that Item-C is an original Item, not a proxy-copy.

But it might not be necessary to have a special "copied item" attribute. We might be able to store everything we need to know just by using the UID. Each UID has two halves: the id of a repository, and the local id of the item within the repository. It may be the case that original Items **always** live their entire lives in the repository where they were first created. If that's true, then we know:

- (a) an Item is an original Item if its uid has a repository id that matches the repository the Item is in now
- (b) an Item is a proxy-copy if its uid has a repository id that does not match the repository the Item is in now
- (c) if an Item is a proxy-copy, then the original Item will have the same uid, and the original Item will be in the repository specified by the repository id in the uid

See also DbOidCoinageProblem.

Partial proxy-copies

When an Item is proxy-copied, the copying won't be instantious. In some cases, there will be thousands of Items to copy at once, and the destination repository may have to chunk the requests that it sends to the source repository. And the network connection might go down mid-way through the operation. Or, in other cases, the destination repository may be smaller than the source repository, and Chandler may intentionally decide to copy only a portion of the interesting Items.
In any case, the result is that a repository may sometimes contain only a portion of a data model. An proxy Item in a repository may be an incomplete copy, or may reference objects in remote repositories.

The issues here we're talking about here, partial proxy-copies in repository, is related to a similar issue, that of having only partial items loaded into client cache memory. For some discussion of that related issue, see:

- DbWheelMetaphorLayerCakeMarch2003
- OsafMarch2003DatabasePlan#ZodbCaching
- OsafMarch2003DatabasePlan#ZodbPartial

Data model requirements

The representation of an Item in a repository must be expressive enough to support the semantics of copies. Here are some of the requirements for proxy-copies.

- **attributes can reference remote items**
  - A local item may have an attribute that points to a remote item. In this case there are no copies.
  - example: My calendar includes a reference to an event from your calendar, and that event is stored in your repository.

- **copies always know what kind they are**
  - a local copy always knows what Kind-of-Item it is an instance of
  - example: My calendar includes a copy of an event from your calendar. The copy knows that it's an Event.

- **copies always know their uid**
  - A local copy always has its uid attribute, and the value is always correct and complete.
  - The uid of the local copy is always identical to the uid of the original object.

- **copies always reference their original remote items**
  - A local item may be a copy of a remote item. The local copy will have a reference to the remote item.
  - example: My calendar includes a copy of an event from your calendar, and the copy includes a reference to the remote item.

- **copies always know that they are copies**
  - A local copy always knows that its a copy.
  - An original always knows that its an original.

- **copies can contain copies of attribute values**
  - A local copy may be a complete copy of a remote copy.
  - example: My copy of your event includes copies all the values of all the attributes.

- **copies can contain missing attribute values**
  - A local copy may be a partial copy of a remote copy. Some of the attribute values have been copied, but other attribute values have not been.
  - example: My copy of your event includes a complete list of the event attributes, and a copy of the value for the "headline" attribute, but does not include copies of attribute values for all the other attributes.

- **copies can have missing attribute lists**
  - A local copy may not even know the names of its attributes, much less their values.
  - example: my copy of your event is a completely empty proxy, which just points to your event

Strawman solution

Here is one mechanism that might be adequate enough meet the needs outlined in the list of requirements above:

- **three manditory attributes**
  - All instances have three mandiatory attributes. All original instances have these attributes, and all copies have these attributes.
    - uid
    - instance of
    - attribute list

- All instances have valid attribute values for the "uid" and "instance of" attributes.
- For the value of the "attribute list" attribute:
  - In orginal instances, the value of the "attribute list" attribute is a dictionary of attributes and attribute values
  - In copies with missing attribute lists, the value of the "attribute list" attribute is the special value UNCOPIED
  - In copies with missing attribute values, the value of the "attribute list" attribute is a dictionary of of attributes and attribute values, in which the attribute values may be special value UNCOPIED. Having an UNCOPIED attribute value is different from having a NULL attribute value.

Here are a few examples:

original Event (in repository A)		
uid	A-349	
instance of	Event	
attributes	"headline"	"Lunch with Tug"
	"start time"	1:30pm 15 Mar 1994
	"end time"	2:30pm 15 Mar 1994
	"notes"	"bring March report"

local proxy (in repository B)	
uid	A-349
instance of	Event
attributes	UNCOPIED

local proxy (in repository C)		
uid	A-349	
instance of	Event	
attributes	"headline"	"Lunch with Tug"
	"start time"	1:30pm 15 Mar 1994
	"end time"	2:30pm 15 Mar 1994
	"notes"	UNCOPIED

local proxy (in repository D)		
uid	A-349	
instance of	Event	
attributes	"headline"	"Lunch with Tug"
	"start time"	1:30pm 15 Mar 1994
	"end time"	2:30pm 15 Mar 1994
	"notes"	"bring March report"

Proxy-copies of non-Chandler data

All of the above discussion focused on the case where one Chandler repository has proxy-copies of the Items that live in another Chandler repository. Alternatively, a Chandler repository may have proxy-copies of data that lives in non-Chandler

repositories. For example, a Chandler repository might have a local proxy-copy Event that points to an original "Event" record on a "foreign" CAP server.

So then, how should those proxy-copies be represented in the Chandler data model? Here's one option... We could create a new Kind-of-Item, that would represent a reference to a foreign record. Let's call that a Foreign-Item. A Foreign-Item would have attributes that could be used to point to records on foreign servers. A proxy-copy would have an attribute, "proxy for", that would point to an instance of a Foreign-Item, and then other attributes of the proxy-copy might be filled in, or might have the value UNCOPIED.

Here are a few examples.

original VEVENT data (on a CAP server)	
"UID"	19970901T130000Z-123401@host.com
"SUMMARY"	"Annual Employee Review"
"DTSTART"	19970903T163000Z
"DTEND"	19970903T190000Z

Foreign-Item (in repository E)		
uid	E-596	
instance of	Foreign-Item	
attributes	"CAP server"	"host.com"
	"UID"	"19970901T130000Z-123401"

local proxy (in repository E)	
uid	E-852
instance of	Event
proxy for	E-596
attributes	UNCOPIED

Foreign-Item (in repository F)		
uid	F-128	
instance of	Foreign-Item	
attributes	"CAP server"	"host.com"
	"UID"	"19970901T130000Z-123401"

local proxy (in repository F)		
uid	F-769	
instance of	Event	
proxy for	F-128	
attributes	"headline"	"Annual Employee Review"
	"start time"	UNCOPIED
	"end time"	UNCOPIED

Foreign-Item (in repository G)		
uid	G-033	
instance of	Foreign-Item	
attributes	"CAP server"	"host.com"
	"UID"	"19970901T130000Z-123401"

local proxy (in repository G)

uid	G-244	
instance of	Event	
proxy for	G-033	
attributes	"headline"	"Annual Employee Review"
	"start time"	4:30pm 03 Oct 1997
	"end time"	7:00pm 03 Oct 1997

Discussion

- We've been talking a lot about partial objects. A use case: In my repository, I have a bunch of contacts, each item has the attributes "name", "phone" and "note". In the note attribute I store personal observations about each contact. I want to share my contacts with you, but I don't want to share the "note" attributes. When you do a query on my repository, you get items with only the "name" and "phone" attributes. Perhaps you don't even know that the "note" attribute exists in my repository. This case is very similar to the case that you describe as "partial proxy-copies". -- KatieCappsParlante, 24 March 2003

...Proposed Strawman Solution...

- Have the data model use a uid in a Chandler proxy copy that is identical to the uid in the Chandler original.
- Have the data model support Chandler proxies using the UNCOPIED value.
- Have the data model support foreign proxies using the "proxy for" attribute.

Text Strings

I'm not sure how text strings should be represented in the data model. There are a number of issues...

Translated strings

Some strings will have different versions for different languages.

Langauge	String
English (en)	Hello World
German (de)	Guten Tag Welt

For example, when a Parcel is localized, the names of different Items in the Parcel might be translated. There might be translations for the name of the Parcel itself, or for the names of Items ("Event, "Contact", etc.), or for the names of attributes ("Start Time", "Name", etc.). For that matter, the list of languages themselves will have different translations ("German", "Deutsch", "Allemand", etc.).

And user-defined Items might also include translated strings. A simple list of Countries or Cities might include different names in different languages.

The translated strings are different versions of each other. They're related to each other, and the data model representation should reflect these relationships. From the data model, for a given string, you should be able to find out what languages the string is available in, and what the value of the string is in each language. The UI should know what language the user wants to work in, and if a String has a translation into that langauge then the UI should present that version of the string. And an end-user should be able to add a new translation to a different langauge, even for an Item they didn't create.

In one of the design list posts, David Neeley talked about the idea of having all items being multilingual-capable, and the more general idea of having different aliases for items.

Non-translatable strings

Most strings should probably be allowed to have translations, but some strings, by definition, should have only a single canonical form. For example, the ISO language codes ("en", "de", etc.) are definitely string literals, but they should not be allowed to have translations. And the same for some country codes, or the abbreviations for US states ("CA", "WA", "TX", etc.). Users should be able to create new Items with attributes that are specified as being non-translatable strings.

One-liners vs. text blocks

Some strings are one-liners, and some are text blocks. For example, in an address book, the "name" of an company should be a one-liner, without tabs or carriage returns, whereas a "directions" field can be a text block, with tabs or carriage returns. The data model needs to be able to represent the fact that some strings should be restricted to one-liners.

ASCII vs. rich text

Some strings are plain ASCII text. ***Other*** strings have rich text formatting, like **bold** and *italics* and links. One-liners can sometimes have rich text, and text blocks can sometimes be restricted to plain ASCII text.

The data model needs to be able to represent the fact that some strings should be restricted to plain ASCII text. And for rich-text strings, the data model needs to be able to represent the rich text, although probably that can be done with simple in-line markup, so that the data model doesn't need to know much about it.

Maybe Chandler can get away with only having a single representation for rich text, but it's possible that Chandler will need to import text in a variety of formats, like HTML, or RTF. The data model may need to know about these different formats, so that for a given string the data model knows which format the string is in.

Representation

I'm not sure how all of this should be represented in the data model. There will be a lot of strings in a repository, so efficiency is important. Simple strings should be represented simply. But it may be that complicated strings should be represented as first-class Items, so that a user could look at a table of translations in a generic table

view, or use a generic query language to see information about a collection of strings. And it may be that a user should be able to dynamically change the schema, so that a simple type of string can become a more complicated type of string.

Anyway, all this deserves some thought. For now I've basically punted, and this strawman data model has a fairly naive set of representations for string types.

...Proposed Strawman Solution...

- Have the data model support string values as literals, for simple strings and for non-translatable string codes.
- Have the data model support Language items that represent different languages, with names and ISO codes.
- Have the data model support String items that have associated Langauge items.
- Have the data model support table of Strings items, with translations in different languages.
- Have the data model support text blocks as well as one-line strings.

Compound Attributes

Some items are more complex than literal, but yet still simple enough that they don't need to be first-class, heavyweight Items.

@@@ -- Example: Timespan (or Daterange), Attribute Value,

Why aren't Compound Attributes just types of Items?

- A Compound Attribute doesn't need to have its own UID. A Compound Attribute is wholly owned by a single Item. It is never referred to by any other Item. Furthermore, it is wholly owned by a single attribute of a single Item. It cannot be pointed to by more than one attribute, nor can it appear in more than one collection, or more than once on a list.
- A Compound Attribute doesn't need to have any of the attributes that Items can have. Compound Attribute can never be proxies, and they don't need to have creation times, and version histories, and annotations.

Why aren't Compound Attribute just simple literals?

- @@@

Why aren't Compound Attribute just kept as independent literals?

- @@@

See March 2003 DB Plan: Opaque Pickles

...Proposed Strawman Solution...

- @@@

Repository Subsets

Background

An excerpt from a design list post by Rys:

Another big question is: what is a subset of a repository? ... I keep asking people how we intend to define this for the sake of being able to synchronize only parts of our repository, because synchronizing all of a repository will cause breakage if the repository must contain both shared data and data that should be visible locally only (perhaps because it represents to-do list information in a local client).

(Data hidden for security reasons would work as a means of reducing scope of what gets shared in synchronization, but that would be the wrong way to do it, since it would conflate security mechanisms with synchronization policies.)
But no one has an answer, so perhaps I am to make something up.

Alternatively, data with different purposes might be partitioned by using multiple repositories. But no one really wants to talk about the issues of actually maintaining multiple repositories either. But that seems like something easy to invent on-the-fly when it matters.

And David Paigen offers some answers to these questions in another design list post. Here are some excerpts:

The ideal (a self contained subset of the data&metadata) is not possible because our knowledge is too interconnected. A subset large enough to be useful (e.g. more than a post-it note) will have references to data outside the subset. :-(

I think the network of connections within the data needs to account for loose ends, links that refer to data that is not contained in the local repository and may not be available at all. But this begs the question, how do you know when to include more data into a subset and when to cut the link and leave a loose end? Does this mean that you would have to copy a subset into new memory to operate on it?

I see security as a subset of subset partitioning. The first partition I would want to define on my repository is what I will share and what I don't want to share. I may even slice the data several ways and share work stuff with my boss, personal stuff with my partner, and hobby stuff with a friend.
Partitioning into separate files might be a good answer. Especially if Chandler can present them to me as seamlessly stitched together. Or mostly seamless, anyway. :-)

There would need to be some sort of partition tool, where you begin by dropping some items into a partition (or via a query), and the tool would explore the connections and allow you to choose which connected data and metadata to add to that partition.

And there would have to be some way of re-inserting a partition into a repository, some way of restoring those loose end links.

Options

Maybe create a Kind-of-Item named something like "Partition" or "Project" or "Workspace". Most users would only ever want to have a single Workspace, but if people wanted to they could create separate Workspaces. For example, if I'm a lawyer or an independent consultant, I might have one Workspace for each of my clients. That way I can make independent back-ups of each workspace, and I know that I won't commingle the proprietary documents of one client with those of another.

...Proposed Strawman Solution...

- @@@

Division of Labor

Background

An excerpt from a design list post by Rys:

here is a question which comes up repeatedly in our designs:

Does the database do it, or does the application do it?

For example, in the case of synchronization and logged changes, does the database guarantee (or seem to, in the interface) efficient access to logged changes so they can be traversed directly? Until an API for this appears in RAP (the repository access protocol), the answer is no.

Or does the application make sure relevant content is marked with timing info, so that change time can be indexed, allowing efficient traversal by virtue of an index. Until we design an API that specifies index behavior in a repository, and then reveal this in rap, the answer is no.

Middleware

Other Chandler architecture documents have also talked about a middleware layer, and about how functionality is divided up between the client code, the middleware code, and the server code.

Implications for the data model

Those decisions, about what functionality is implemented where, may end up having implications for the design of the data model. Here's a list of some of the areas of Chandler functionality, and notes about the data model implications of having that functionality in one place versus another.

- timing info: creator, creation time, las modified time, etc.
    - implications: @@@
- mapping between change objects and item versions
    - implications: @@@
- persistent event queues
    - implications: @@@
- observable queries
    - implications: @@@
- synchronization and replication
-

...Proposed Strawman Solution...

- @@@

Proliferation of Items

The original descriptions of Chandler talked about a variety of different kinds of Items, like Events, Email Messages, Contacts, Tasks, and Notes. And in reading through the wiki pages, it becomes clear there there need to be lots of other kinds of Items too, like Personas, and Groups, and Notifications. But it's not clear how far all of this should go. How do you decide what should be an Item, and what shouldn't be?

Views as Items

I understand that the Chandler team has talked some about the pros and cons of treating Views as just another type of Item. I'm entering the conversation a little late, so I don't know what ground has already been covered, or what the current consensus. But, for a glimpse of some background, see DB Topics: View Model.
From my perspective, approaching things from a data-modeling point of view, it seems very clear to me that Views should be treated as just another type of Item. That users can leverage all of Chandler's powerful features in working with the Views themselves. If Views are just Items, then users can query repositories to find out about available Views, and see Views that show lists of Views, and publish lists of Views, and subscribe to Views of Views, and add annotations to their friend's Views, and collaborate to create better Views.

See also: Mike C. Fletcher's design list post about "View definitions as items in the the database".

For performance reasons, it's probably necessary to have a client-side cache of the View meta-data. But that shouldn't be a problem. Client-side caching and off-line work are already clear architectural goals.
So, for now, in this strawman data model I've gone ahead on the assumption that Views are Items.

Meta-data as Items

Just like the question about whether Views should be first-class Items, there's a similar question about all the meta-data. Should attributes be Items? Should a Kind-of-Item be an Item?

I'm sure there are good pro and con arguments, but I haven't talked those through with anyone, and I'm not up on the current OSAF thinking. From my perspective, it seems like it'd be great if all the meta-data was just a bunch of Items. And then you get all the Item-centric benefits of Chandler: the ability to query it, to see it displayed in generic table views, to share those views with other people, etc.

Everything else as Items

Along with the Views and the meta-data, there are dozens and dozens of other objects that could be modeled as Items. Things like Queries, Triggers, Preferences, Timezones, Resources, and Languages. And on and on. Almost across the board, I've defaulted to treating all these things as just other kinds of Items. I don't know what the current OSAF thinking is about all this, and I'm worried that I may have wandered far off from the current consensus vision, but it seems so **compelling** to just treat everything as an Item. And I'm having a hard time thinking up good reasons not to.

So, with that caveat, here's the list of the hundred-odd Kinds of Items, described in the Data Model Schema.

...Proposed Strawman Solution...

- When in doubt, make it a first-class Item.

Contributors

- KaitlinDuckSherwood - 09 Dec 2002
- BrianDouglasSkinner - 19 Feb 2003
- BrianDouglasSkinner - 17 Mar 2003 to 14 Apr 2003
- BrianDouglasSkinner - 30 May 2003
