The Busy Developer's Guide to the Chandler repository

Version 0.4

This document is an introduction to the features of the Chandler repository. It's not meant to give exhaustive coverage of the entire API or all of the cool features. Instead it is meant help you get up to speed so that you can do common repository tasks.

The Chandler repository allows you to store semi-structured or loosely structured information. It forms the basis for the Chandler PIM application by providing the ability to model, store, and retreive personal information. The repository supports a number of unusual features (described below) that make it a good fit for building a PIM. It can also be used as a stand alone storage system for applications that need to store semi-structured information.

The Chandler repository supports some features which are unusual when compared to most storage systems. The unit of storage in Chandler is the Item. Items are interesting in a number of ways. You can extend the schema that describes an Item by adding a new attribute to an Item -- this causes the schema to be updated. Chandler Items can also refer to other Chandler Items. In most object like storage systems references are unidirectional, like pointers. In the Chandler data model, references can be bidirectional -- each end of the reference is aware of the other end.

Another unusual feature of the Chandler repository is the concurrency model. Chandler uses an optimistic concurrency control mechanism based on versions of Items. If you are familiar with CVS, you will find some conceptual similarities. The biggest thing to be aware of is that commits can fail due to version conflicts, and that your application is responsible for recovering from these failures.

Chandler Data Model

Chandler's repository stores and retrieves persistent objects called "Items"; these Items are arranged in a hierarchy, and are addressable either by their unique identifier (UUID) or by their path within the hierarchy (e.g. //Schema/Core/Kind). There are special Items called "Kinds" which play the role of classes in the object-oriented world. Items can be thought of as instances of Kinds, and take on the characteristics described by their Kind. For example, a Kind determines what attributes an Item has and what code implements an Item's behavior. An attribute contains meta information including its name (how you refer to it in Python), cardinality (single or multi-valued) and type (what kind of value can be assigned).

There are two ways to create Items in the repository: loading them in from XML files known as "parcels", or by using the repository API directly.

Parcel Loading

A parcel is a set of Items (defined in an XML file) to be loaded into the repository and, optionally, code implementing custom behavior for those Items. When Chandler starts up, the Chandler/parcels directory is recursively searched for "parcel.xml" files which are then parsed by the parcel loader. The Items for a parcel are loaded into the repository path //parcels, using a path determined by the parcel's location on disk. For example, a parcel being loaded from Chandler/parcels/OSAF/contentmodel/calendar/parcel.xml will be stored in the repository at the path //parcels/OSAF/contentmodel/calendar.

Defining Schema

Let's look at a simple parcel from the repository unit tests, chandler/application/tests/testparcels/simple/parcel.xml:

```
<Parcel itsName="simple"
       xmlns="http://osafoundation.org/parcels/core"
       xmlns:simple="http://testparcels.org/simple" >

  <namespace value="http://testparcels.org/simple" />

  <displayName value="Simple Parcel" />
  <description value="Simple Parcel Loader Test Schema" />
  <version value="0.1" />
  <author value="Open Source Applications Foundation" />

  <Kind itsName="TestKind">
    <displayName value="Test Kind" />
    <attributes itemref="simple:TestAttribute"/>
    <attributes itemref="simple:ListAttribute"/>
    <attributes itemref="simple:DictAttribute"/>
    <displayAttribute itemref="simple:TestAttribute"/>
  </Kind>

  <Kind itsName="SubKind">
    <superKinds itemref="simple:TestKind"/>
    <displayName value="Subclass Test Kind" />
  </Kind>

  <Attribute itsName="TestAttribute">
    <displayName value="Test Attribute" />
    <cardinality value="single" />
    <type itemref="String"/>
    <initialValue type="String" value="XYZZY" />
  </Attribute>

  <Attribute itsName="ListAttribute">
```

```
    <displayName value="List Attribute" />
    <cardinality value="list" />
    <type itemref="String"/>
    <initialValue/>
  </Attribute>

  <Attribute itsName="DictAttribute">
    <displayName value="Dict Attribute" />
    <cardinality value="dict" />
    <type itemref="String"/>
    <initialValue/>
  </Attribute>

</Parcel>
```

As the parcel loader parses the XML, whenever it sees an element with the itsName attribute it creates an Item in the repository. For a parcel file, the outermost element should be Parcel. When defining Kinds and attributes, use elements of Kind and Attribute. Remember that all elements of these types must be in the XML namespace http://osafoundation.org/parcels/core. The nesting arrangement of the Items in a parcel file will be duplicated in the repository hierarchy.

The <namespace> element tells the parcel manager which namespace should be associated with this parcel. Note that we are using a dummy namespace URI "http://testparcels.org/simple" in this example. You should use a meaningful namespace URI for your parcels.

Now let's look at the <Parcel> element: its itsName attribute is used to name the Item in the repository. By convention a parcel Item's name needs to match the directory in which the parcel.xml file resides (in this case, "simple"). The parcel loader will create an Item of Kind "//Schema/Core/Parcel" in the repository, with a path of //parcels/simple.

When the parcel loader encounters an element that does not have an itsName, the element is instead used to assign a value to one of the parent element's attributes. In the example, the parcel's displayName attribute is assigned the literal "Simple Parcel". Another type of assignment is a reference; if an element has an itemref, then instead of a literal assignment, a reference to the specified Item is assigned. For example, within the "TestKind" Item, a reference to the "TestAttribute" Item (also defined in this file) is assigned to TestKind's "attributes" attribute.

The result of parsing this file will be:

- A parcel Item at //parcels/simple (with displayName="Simple Parcel")
- A Kind Item at //parcels/simple/TestKind (with displayName = "Test Kind", attributes = [//parcels/simple/TestAttribute, //parcels/simple/ListAttribute, //parcels/simple/DictAttribute], and displayAttribute = //parcels/simple/TestAttribute)

- A Kind Item at //parcels/simple/SubKind (with displayName = "Subclass of Test Kind", superKinds = //parcels/simple/TestKind
- An attribute Item at //parcels/simple/TestAttribute (with displayName = "Test Attribute", cardinality = "single", and type = //Schema/Core/String)
- An attribute Item at //parcels/simple/ListAttribute (with displayName = "List Attribute", cardinality = "list", and type = //Schema/Core/String)
- An attribute Item at //parcels/simple/DictAttribute (with displayName = "Dict Attribute", cardinality = "dict", and type = //Schema/Core/String)

So what does this mean in terms of the data model?

1. Our simple schema has defined two Kinds (TestKind and SubKind) and an attribute (TestAttribute)
2. We have declared that instances of TestKind (Items whose Kind is TestKind) have 3 attributes named "TestAttribute", "ListAttribute", and "DictAttribute" (by assigning to the "attributes" attribute)
3. We have declared that SubKind is a "subclass" of TestKind (by assigning to the "superKinds" attribute). SubKind will inherit the characteristics of TestKind, including the list of allowed attributes.

Defining Data

Let's create some data based on this schema. As of the 0.3 release, you need to put your data (non schema) Items in a separate parcel file from your schema, so we've put this in Chandler/repository/parcel/tests/testparcels/simple/data/parcel.xml:

```
<Parcel itsName="data"
      xmlns="http://osafoundation.org/parcels/core"
      xmlns:simple="http://testparcels.org/simple" >

 <namespace value="http://testparcels.org/simple/data" />

 <simple:TestKind itsName="item1">
  <TestAttribute>xyzzy</TestAttribute>
 </simple:TestKind>

 <simple:SubKind itsName="item2">
  <TestAttribute>plugh</TestAttribute>
 </simple:SubKind>

 <simple:TestKind itsName="item3"/>

</Parcel>
```

When this parcel is loaded, three new Items will be added to the repository:

- A parcel Item at //parcels/simple/data
- A TestKind Item at //parcels/simple/data/item1 (with TestAttribute="xyzzy")

- A SubKind Item at //parcels/simple/data/item2 (with TestAttribute="plugh")
- A TestKind Item at //parcels/simple/data/item3 (with TestAttribute="xyzzy" - because the initialValue for TestAttribute is "xyzzy")

We were able to assign to the "TestAttribute" for item1 because we declared it was an attribute of TestKind. Since the data model honors Kind inheritence, all instances of SubKind also have the "TestAttribute" attribute, and therefore we could assign it to item2 as well.

Using the Repository API

Now that we've covered how parcels can be used to populate the repository, let's interact with them via the repository API.

Getting Started

The first thing that we are going to need is a repository object that we can work with. When you are working within Chandler, there is an application global, application.Globals.repository whose value is the repository that the current Chandler instance is using. I'm going to show you how to obtain a repository object from scratch.

Use the Chandler build tool, hardhat, to run an interactive python session. You'll want to be in the "chandler" directory when you do this.

```
> cd osaf/chandler
> ../hardhat/hardhat.py -i
Python 2.3.3 (#1, Sep 22 2004, 23:52:03)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1495)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

We'll be using this throughout the rest of the guide.

First we need some modules from Chandler

```
import os
from repository.persistence.XMLRepository import XMLRepository
```

Next we need to tell where to find the repository. The Chandler repository stores its data in a directory somewhere on your computer's filesystem. As a convention, the repository's directory is named __repository__. Let's create a brand new repository in the current directory, using the XMLRepository constructor, which needs the path for the repository directory.

```
rep = XMLRepository(os.path.join('.', '__repository__'))
```

Once we have a repository object we can call the create method to actually create the repository on disk.:

```
rep.create()
```

The XMLRepository.open() method will create a repository if there isn't one.

The next three lines load the default Chandler schema, which we'll be using for examples.

```
schemaPack = os.path.join('repository', 'packs', 'schema.pack')
rep.loadPack(schemaPack)
rep.commit()
```

Loading Parcels

Next let's load some example parcels from the unit tests. Normally as part of the Chandler startup process, all parcels in osaf/chandler/Chandler/parcels are loaded automatically. For this interactive session, however, we'll just explicitly load our data parcel instead:

```
from application.Parcel import Manager as ParcelManager

manager = ParcelManager.getManager(rep, ['./application/tests/testparcels', './parcels'])

namespaces = ['http://testparcels.org/simple/data']
manager.loadParcels(namespaces)
```

The first line imports the packages we need in order to use the Parcel loader. We call the methodParcelManager.getManager in order to get an instance of the Parcel loader. The first argument, rep, is the repository that we've just created. The second argument is a list of directories to search for parcels. The first element in the list is the directory containing the parcel.xml file for our data parcel. The second element is the root of the OSAF supplied parcels. The list namespaces is a list of namespaces (URI's) for the parcels that you want to load. In this case, our data parcel depends on our schema parcel, and so the schema will automatically be loaded as well.

To see what's in the repository, we can print out part of its contents using PrintItem( ):

```
from application.Parcel import PrintItem as PrintItem
PrintItem("//parcels/simple", rep)
```

You should get something like this, a description of //parcels/simple and all of it's children (recursive):

```
//parcels/simple (Kind: //Schema/Core/Parcel)
   author: Open Source Applications Foundation <type 'unicode'>
   createdOn: 2004-10-07 17:40:41.01 <type 'DateTime'>
   description: Simple Parcel Loader Test Schema <type 'unicode'>
   displayName: Simple Parcel <type 'unicode'>
```

```
  file: ./application/tests/testparcels/simple/parcel.xml <type 'str'>
  modifiedOn: 2004-10-07 17:40:41.01 <type 'DateTime'>
  namespace: http://testparcels.org/simple <type 'unicode'>
  namespaceMap: (dict)
  originalValues: (dict)
    : {'namespace': u'http://testparcels.org/simple',
       'version': u'0.1',
       'displayName': u'Simple Parcel',
       'description': u'Simple Parcel Loader Test Schema',
       'author': u'Open Source Applications Foundation'}
    ListAttribute: {'displayName': u'List Attribute',
                    'cardinality': u'list',
                    'initialValue': '',
                    'type': 'ab8c18ea-18c2-11d9-85f4-000a959a114e'}
    TestAttribute: {'displayName': u'Test Attribute',
                    'cardinality': u'single',
                    'initialValue': u'XYZZY',
                    'type': 'ab8c18ea-18c2-11d9-85f4-000a959a114e'}
    TestKind: {'attributes': ['ae479b5e-18c2-11d9-85f4-000a959a114e',
                              'ae4b2652-18c2-11d9-85f4-000a959a114e',
                              'ae4c3920-18c2-11d9-85f4-000a959a114e'],
               'displayName': u'Test Kind',
               'displayAttribute': 'ae479b5e-18c2-11d9-85f4-000a959a114e'}
    SubKind: {'superKinds': ['ae3f7c80-18c2-11d9-85f4-000a959a114e'],
              'displayName': u'Subclass Test Kind'}
    DictAttribute: {'displayName': u'Dict Attribute',
                    'cardinality': u'dict',
                    'initialValue': '',
                    'type': 'ab8c18ea-18c2-11d9-85f4-000a959a114e'}
  version: 0.1 <type 'unicode'>
```

Note the inclusion of UUID's (strings that like 'ab8c18ea-18c2-11d9-85f4-000a959a114e').

Working with Items

One way to retrieve an Item from the repository is to ask for it by its path. So to fetch item1:

```
item1 = rep.findPath("//parcels/simple/data/item1")
```

If you print item1

You'll get something that looks like this:

```
<Item: item1 843636cc-55bd-11d8-f385-000a95bb2738>
```

This shows the Item's class name (this may change to be the Kind), name, and UUID. Every Item in the repository has a unique identifier or UUID. You can use the itsUUID attribute to get an Item's UUID.

```
uuid = item1.itsUUID
```

You can use the findUUID method to find by using an Item's UUID.

```
item2 = rep.findUUID(uuid)
```

Note: at the moment, you should be careful when using UUID's to find Items, because UUID's really are unique, are never recycled, and it's possible for you to use a UUID that existed in some older version of your repository, but which doesn't exist anymore in the current repository if it was recreated.

Once you have an Item, there are some basic things that you can do with it. We've already seen that you can get the UUID using getUUID(). Here are some other things that you can do:

Item.itsName	get the name of the Item
Item.itsParent	get the Item which is the parent of this Item
Item.itsPath	get the path to this Item
Item.getItemDisplayName()	get the displayable name for this Item

Note that in our example, item1.getItemDisplayName() will evaluate to "xyzzy". Why? In our schema we defined TestAttribute to be TestKind's "displayAttribute", therefore that attribute's value is returned by getItemDisplayName(). If an Item's Kind does not have a displayAttribute, then the Item's name will be returned instead.

Parents and children

As you've seen in these examples, Items can be arranged in a parent/child hierarchy. This can be useful for related groups of Items. When you create an Item, you specify what its parent will be. Items can have the repository as their parent.

Let's look at adding a new instance of TestKind. To do this, we first need to retrieve the TestKind Item:

```
testKind = rep.findPath("//parcels/simple/TestKind")
```

Next we need to fetch the Item that will be our new Item's parent:

```
data = rep.findPath("//parcels/simple/data")
```

Next we tell the Kind to create a new Item as a child of data:

```
newItem = testKind.newItem("newItem", data)
```

The first argument to the newItem() method is the name of the Item, and the second argument is the parent of the Item.

You can test whether or not an Item has any children
```
print data.hasChildren()
```

You can iterate through the children of an Item

```
for child in data.iterChildren():
    print child
```

You can test whether or not an Item has a child with a particular name

```
print data.hasChild("newItem")
```

You can get a child by providing its name. There are 2 ways of doing this. The first way returns None if the child is not found:

```
print data.getItemChild("newItem")
```
The second way raises a KeyError exception instead):
```
print data['newItem']
```

You can rearrange the children of an Item by using the placeChild() method.

```
data.placeChild(data.getItemChild("newItem"), None)
```

will make the 'newItem' child the first child of data. The first argument is the child you want to move, the second argument is the child that you want to place the first child after. You supply None to place the child as the first child.

To change an Item's parent you would use the move() method. If I wanted to take 'newItem' and make it a child of the repository, the code would look like this:

```
newItem.move(rep)
```

The first argument to move is the new root for the Item being moved. You can also specify where the Item will be among the new root's children by specifying a 2nd and 3rd argument, which specify the children preceding and following the new Item.

Attributes

All of the data in Chandler Items is stored as attributes. Attributes are themselves Items, albeit of a special type (the recursion has to stop somewhere). The API for attributes has been designed to look as much like the Python API for working with Python attributes. If we look at the Item we created, newItem we can set its attribute values using the normal Python syntax:
```
newItem.TestAttribute = 'Testing'
```

Sets the value of newItem's *TestAttribute* attribute to Testing. We can retrieve the value of the TestAttribute attribute by executing

```
newItem.TestAttribute
```

Where do attributes come from? In the Chandler data model, the Kind for an Item determines what attributes are available. In the case of the Kind for parcels, there are a number of attributes available. To find out what attributes are available on a Kind, we can look at the attributes attribute of that Kind:

```
parcelKindItem = rep.findPath("//Schema/Core/Parcel")
for i in parcelKindItem.attributes:
    print i
```

Which will produce output that looks like this:

```
<Attribute: author 0114bf18-4cb5-11d8-d2ac-000a959a114e>
<Attribute: publisher 011593c0-4cb5-11d8-d2ac-000a959a114e>
<Attribute: status 0116b7d2-4cb5-11d8-d2ac-000a959a114e>
<Attribute: summary 0117c802-4cb5-11d8-d2ac-000a959a114e>
<Attribute: icon 0118e868-4cb5-11d8-d2ac-000a959a114e>
<Attribute: version 0119e696-4cb5-11d8-d2ac-000a959a114e>
<Attribute: createdOn 011ad7ea-4cb5-11d8-d2ac-000a959a114e>
<Attribute: modifiedOn 011bc1be-4cb5-11d8-d2ac-000a959a114e>
```

However, this list only includes attributes that were directly assigned to the //Schema/Core/Parcel Kind -- this Kind is a Subkind of //Schema/Core/Item and therefore it inherits Item's attributes as well.

There are two Kinds of attributes. Literal valued attributes store literal values of various types, such as numbers (integers, longs, complex), strings, dates, and so forth. Reference valued attributes store bi-directional references to Chandler Items.

Attributes also have a cardinality. They can be single valued or multiple valued. Multiple valued attributes are treated as either lists, accessible by numeric index, or as dictionaries, accessible by keys. Multivalued reference attributes are treated as lists. The Item.iterAttributeValues() method will produce an iterator that will let you access all the attributes for an Item, including thos attributes inherited from Superkinds:

```
parcel = rep.findPath("//parcels/simple")
for i in parcel.iterAttributeValues():
    print i
```

The output looks like this:

```
(u'displayName', u'Simple Parcel')
(u'description', u'Simple Parcel Loader Test Schema')
(u'author', u'Open Source Applications Foundation')
('createdOn', <DateTime object for '2004-02-02 13:34:02.80' at 24b9410>)
(u'version', u'0.1')
```

```
('modifiedOn', <DateTime object for '2004-02-02 13:34:02.80' at 24b9410>)
('kind', <Kind: Parcel 760888e8-55c7-11d8-f88d-000a95bb2738>)
```

You can also use the keyword arguments valuesOnly or referencesOnly to True and iterAttributes() will only produce literal or reference attributes.

```
for i in parcel.iterAttributeValues(valuesOnly=True):
    print i
```

produces:

```
(u'displayName', u'Simple Parcel')
(u'description', u'Simple Parcel Loader Test Schema')
(u'author', u'Open Source Applications Foundation')
('createdOn', <DateTime object for '2004-02-02 13:34:02.80' at 24b9410>)
(u'version', u'0.1')
('modifiedOn', <DateTime object for '2004-02-02 13:34:02.80' at 24b9410>)
```

and

```
for i in parcel.iterAttributeValues(referencesOnly=True):
    print i
```

produces:

```
('kind', <Kind: Parcel 760888e8-55c7-11d8-f88d-000a95bb2738>)
```

You can use Python's list and dictionary operations on multivalued literal attributes. Multivalued references attributes are a different story. Items have an API that allows you to manipulate the values of multi-valued reference attributes. Let's try this with the attributes attribute of parcelKindItem which is a list valued reference attribute.

First we need a UUID to use as the key, so for the sake of example, I'm going to cheat and grab the UUID using the iterator.

```
as = parcelKindItem.attributes.itervalues()
a = as.next()
a = as.next()
print a

<Attribute: publisher 7609f124-55c7-11d8-f88d-000a95bb2738>
Now we'll use a's UUID as a key into the attributes attribute.
print parcelKindItem.getValue('attributes', a.itsUUID)
```

Which produces the expected result:

<Attribute: publisher 7609f124-55c7-11d8-f88d-000a95bb2738>

To add a new Item to the list, all you would do is:

```
parcelKindItem.addValue('attributes',item.itsUUID, item)
```

To remove the value of an Item in the list use removeValue:

```
parcelKindItem.removeValue('attributes',a.itsUUID)
```

You can also use Python operations like del to remove a value from a multi-valued attribute. The append, insert, pop, remove, reverse, sort, and extend operations are supported on list valued attributes, and the update operation is supported on dictionary valued attributes.

Saving data

The whole point of using the repository is that you want your data to become persistent. When you create an Item or update an Item, that Item becomes dirty. You can uses the Item.isDirty() method to see if a particular Item is dirty. You use the commit() method on the repository to make your changed Items persistent. Your code would look something like this:

```
rep.commit()
```

If the word 'commit' makes you think of transactions, that's fine. The Chandler store is transactional, which means that you can atomically commit groups of changes. It also means that transactions are used to manage concurrent access to the repository so that agents and other Python threads can access the repository safely. An important thing for you to know is that it is possible for commits to fail. This will happen when an Item that you want to commit has been modified by someone else between the time that you read it and the time you tried to commit it. When this happens the commit method will raise an exception. At this point it is up to you redo the work in your transaction.

For Reference

You can view the repository API documentation on line. You'll probably want to start with Item, Kind, and Attribute.

To Learn More

We are building out our set of documentation on Chandler. You might be interested in the following:

- The Chandler Query System
- Repository Working Group Home Page

Additional Topics

- SubAttributes
- More about cardinality

- More about references

We want to update and improve this document

Please send any comments to dev@osafoundation.org.

$Revision: 1.12 $
$Date: 2004/11/22 23:28:51 $
$Author: twl $
$Log: repository-intro.html,v $ Revision 1.12 2004/11/22 23:28:51 twl Include links to API docs Revision 1.11 2004/10/21 22:30:39 twl Commit branched docs to trunk Revision 1.9.2.2 2004/10/19 21:19:06 twl Incorporate Ducky's feedback Revision 1.9.2.1 2004/10/18 21:27:03 twl Committing doc changes to branch Revision 1.10 2004/10/15 18:31:04 twl Bugs 2112, 2113 (Doc bugs) Incorporate review feedback Revision 1.9 2004/10/12 20:06:12 twl Fix bug 2113 - update repository overview for 0.4 Revision 1.8 2004/02/28 01:00:13 twl Add motivation paragraphs use osaf.css Revision 1.7 2004/02/24 21:39:28 markie Typos in ms and add keywords at end.