Refactor Packagers to support Chain of Command

Add processing of Content Packages to the submission process – at the point of upload unpack a content package and store the metadata held therein, thereby reducing the need for data entry by the depositor. It is anticipated this will support various METS, SCORM and LOM packages.

Refactoring Packager Framework to support Chaining and Graceful degradation of package formats.

The Chain of Responsiblity pattern is a popular pattern for easing the evaluation of which command in a set would be responsible for execution of a responsibility

Limitations in current Package Ingest framework...

1. Packagers rely on being designated by NAMES from the command-line and or designated by SWORD.
   a. In the face of not knowing the appropriate packager, we would like the repository to offer an option to allow it to guess the packager to use.
2. No graceful degradation exists if the package is just a file or simple zip package
   a. We would like to devise a gracefull degredation when specific "sub-type" can still be determined
3. A series of default Packagers should be able to be constructed that take sane values and use a simple heuristic to determine which packagers can be applied:
   a. If the face of errors in a subtype packager, parent packagers could be set into a debug state, place the item in the submission workflow, allowing the user to review the result prior to to accepting/rejecting.
   b. Treats Submission Workflow as a "Staging tool" prior to pushing content into the repository.

Basic Package Ingester Types

1. FileIngester: Takes any file and stores it in a bitstream, attempts to match BitstreamFormat where possible.
   a. Custom implementations can support format specific tasks (extract metadata of pdf or doc, acquire metadata from mpeg
2. ZipIngester: Takes any Zip file and expands it into attached bitstreams, named by the directory path found in the archive.
   a. Custom implementations may handle different types of archives (ZIP, RAR, TAR, TAR.GZ)
3. ManifestBasedIngester: Takes a Manifest and processes it to embellish the Item and Bitstreams from the manifest file. (Either placed inside a ZIP or standalone)
   a. Custom types to support websites and other types saved as zip packages

Challenges with current METS packager design.

1. The problem is that the current packagers always assume you are processing a manifest file. I simply want to process the contents of the zip and name the item and contents solely on those feathers by default, then build customizaed packagers up based on this. It will make creating new packagers much easier because the basic functionality for getting the bitstreams created will be separated from the evaluation of the manifest.

Making Packagers Smarter

I'd like to write something that uses a chaining or filtering model, where each packager operates like a filter on the content.

Input File > File Ingester > Zip Ingester > Manifest Based Ingester

InputFile > FileIngester > Manifest Based Ingester.

For example, packages that are HTML Learning Object content may have the following configuration in the proposed packager chain:

Controller

- Input File
  - File Ingester
    - Zip Ingester (Processes Packaged Content)
      - HTML Index File Ingester
      - IMSCP Manifest Ingester
      - METS Manifest Ingester
      - LOM Manifest Ingester
      - SoftChalk Manifest Ingester

-
  -
    - XML Ingester (Processes External Content)
      - HTML Index File
      - IMSCP Manifest Ingester
      - METS Manifest Ingester
      - LOM Manifest Ingester
      - SoftChalk Manifest Ingester
      - ATOM ORE Manifest Ingester

In each stage, the PackageIngester would be responsible for either creating, altering or adding detail to bitstreams and metadata for one or more items, if the matching criteria for the stage are met, then the bitstreams or metadata changes associated with the Item are derived from that stage, of course, one need to be careful to map the Zip file contents correctly, so that they will populated as the file names for the resources in the Item.

- FileIngester: Detects Mimetype and maps to appropriate packager (RegisteredBitstreamFormatBasedIngester vs PdfIngester vs ... vs ZipIngester),
- ZipIngester: Processes Contents of Zip File into one or more Bitstreams attached to one or more Items named after the path of the file in the archive.
- HTMLIndexFileIngester: looks for index.* file in root directory and attempts to parse html/head/title and html/head/meta from it.
- IMSCPManifestIngester: Evaluates IMSCP Manifest to verify contents, bundle and naming of Bitstreams (including manifest itself), assign additional metadata to Item
- METSManifestIngester: Evaluates METS Manifest to verify contents, bundle and naming of Bitstreams (including manifest itself), assign additional metadata to Item
- SoftChalkManifestIngester: Evaluates SoftChalk Manifest to verify contents, bundle and naming of Bitstreams (including manifest itself), assign additional metadata to Item

This approach would allow us to, via SWORD or CLI packager, ignore mapping to a specific packager and instead allow the system to detect the appropriate initial packagers to apply based solely on mimetype and then apply further packagers based on zip package contents.

Of course, we would leave the legacy capability in place to allow you to submit just a manifest. But we can extend the case to apply to the "XML" mimetype as well.

- Input File
    - File Ingester
        - XML Ingester
            - IMSCP Manifest Ingester
                - METS Manifest Ingester
                    - LOM Manifest Ingester
                        - SoftChalk Manifest Ingester
                            - ATOM ORE Manifest Ingester

Where the additional Ingesters simply continue to attempt to process the XML file and acquire the further content from the package specific references.

The ultimate outcome will be a packaging system that requires far less policing by the enduser doing the ingesting. If it were the case that SOWRD could be parameterized to leave the resulting submission in the users submission queue, they could also review it for appropriate state without formally submitting it to the collection for review/archive, even allowing them to delete the submission so they can repeatedly test the outcome until they get their configuration correct.

Because the ingest chain fails gracefully, they know at which stage the failure is occurring at, can review the state of the current item and can review their SWORD output to determine what when wrong in that packager.

Use Cases : Areas this will be useful in

Submission Workflow

Adding Packager Processing into the Submission File Upload Step so that an Item can be easily viewed and adjusted after being processed at upload. Example Requirements:

1. Upload Step
    a. Initially Processes file or package into Item + Bitstreams, attempts to determine optimal packager type
    b. user is offered selection of processing options.
    c. based on choice, packager is executed
        i. for instance if PDF metadata is available, it is inserted into Item metadata
        ii. for instance if Softchalk Package is detected, index.html or other files are processed to attach Item Title and or other details extracted from source.
2. Edit Metadata Step, give the user a chance to review and add metadata, required fields that may have been absent on upload can now be added.
3. CC License Step, if License we detected in Upload stage, its now attached to the Item and can be viewed.
4. Final Repo License Step
5. Final review and Submission