Production Framework - 20170117

Risk assessment - including mitigation of risks for staying with Drupal longterm - see spreadsheet work (Risk Analysis)

**RECOMMENDATION:**

Aim to replace reliance on Drupal post prototype delivery (June 2018)

Start work on testing COGS as soon as possible.

Preconditions:

Resolution of outstanding issues:

* Development of interface for straight-forward entry of classes, properties, relationships etc without losing the batch entry (CSV) entry option
* Decision on workflow
	+ XMI as canonical point for generation of downstream outputs
	+ CSV based model as canonical point for generation of downstream outputs
* Validation framework - incompatible duplicates, orphan identification etc
* Business rule validation - e.g modelling guidelines
* Migration plan (e.g diff COGS output vs Drupal output)
* Support for bulk editing e.g across more than one class

Finalise testing by end of May 2018 for smooth transition.

**Production Framework for Prototype:**

The following needs to be supported by Drupal in the short-term for production of the June 2018 prototype:

* Critical issues
	+ Validation of Views business rules (re: orphans, inclusion of Complex Data Types, all classes are from packages included in build)
	+ Set status flags values for classes so that those that are prioritized, reviewed, and approved could be appropriately flagged
	+ Running Sphinx in production workflow (Jenkins)
* Non-critical issues:
	+ Integration of examples into Drupal
	+ Formating of descriptive content

This process presupposes that coverage of DDI4 Prototype will be defined, priorities set, and work focused on those priorities over the next 8-9 months.

Package as means of organizing the Library of Classes

* Is the purpose for management - differentiation of core from extensions
* Relating major areas ala GSIM
* Easy viewing of small easy to view subsets
* Separation of types of classes: pattern, complex data types, primitives, realizations of patterns

Views as a means of informing implementers and entry users what classes are used for specific use cases

* Direct users to certain parts of the model for specific purposes
* Arbitrary selection of classes from the library
* Regardless of how created requires examples and use case specific documentation

Technical issues can/should be addressed further downstream in terms of queries etc.

Best assistance for users (content entry) and implementers:

* Examples - use case specific
* Graphical images
* Context sensitive documentation down to class level - use case specific

This implies a set of metadata (version agnostic) that can be used to generate examples in each version and binding (noting what cannot be handled by a specific version or binding). It would show not only the comparability between versions but aid in the selection of a version to accomplish specified goals of the use case.