# Attribute Template Design Approach (Draft)

An attribute template is collection of attributes and their associated layouts, constraints, default values, and other metadata. They are meant to be a generalization of the functionality provided by type definitions in order to re-use that infrastructure in support for classification as defined in PartsLink CP ####: “Classification”.

Attributes templates associated with the type of an object apply to all instances of that type. Under this new approach, additional attribute templates can be defined (outside of the Type Manager) which apply to instances of types based on the value of a particular binding attribute. I.e., attribute template X applies to all parts where the ‘color’ attribute has a value of ‘RED’. Or, attribute template Y applies to all documents where the ‘document classification’ attribute has a value of ‘ABC’.

The proposed approach is to enhance the existing typing infrastructure to store additional hierarchical attribute templates that are not associated with types, to include the information required to differentiate between attribute templates used for typing and those used for other purposes and to support nested attribute layouts in the end user UIs. This will allow the layouts from any attribute templates associated with an object to be included as a subset of the overall layout defined for a type.

**Model**

The changes consist of:

* An 'attTemplateFlavor' integer field would be added to LWCTypeDefinition. The 'attTemplateFlavor' field would be used to indicate whether the LWCTypeDefinition is an attribute template or a full-fledged type definition. (Note: In the final design, this may instead be done with a subclass or an abstract super class.)
* A new 'LWCNestedTemplateGroupDefinition' subclass of LWCGroupDefinition with an attribute reference field would be created. This form of group would be used in layouts for type definitions to indicate the position where an attribute template’s layout information should be inserted into the type’s layout, and it would specify the binding attribute used to determine whether or not that attribute template applies.

**Storage**

Attribute template information will be stored and cached along with existing type definitions.

Standard attributes and Alias attributes will only be supported on full-fledged type definitions, not on attribute templates.

**Administration UIs**

The type manager client will support creation of nested template groups in layouts, allowing the admin user to specify the binding attribute associated with the attribute templates. This is roughly analogous to the existing batch attribute support through dynamic attribute groups.

**End User UIs**

When UI's encounter an LWCNestedTemplateGroupDefinition in a type's layout, they will retrieve the layout info for the attribute template referred to by the attribute the group references. This will be done by passing the attribute and its value for the given object to a dedicated service which will lookup and return the appropriate attribute template for that specific attribute/value pair.

When the value of a binding attribute is modified, the end user UI will delete all attribute values from the old attribute template(s) that no longer apply in the new attribute template(s), and re-layout the UI according to the new attribute template(s).

**Concerns/Risks**

Overall performance, especially memory usage, of retrieving and caching the attribute templates is a potential concern. If the number of attribute templates is significant, the cache size and memory usage will need to be monitored carefully. Performance of reading the attribute templates from the db into the cache is also a potential concern.

There will effectively be a two pass rendering. One to retrieve and render the layout and data associated with a type, and a second to retrieve and render the layout and data associated with any attribute templates included in the outer type layout.

**Expected Changes Required in Existing Typing Infrastructure**

* Update type view, import/export, factory, and db service code to support the model changes.
* Update the type “synchronizationn” and other type specific behaviors in the infrastructure to only trigger on full-fledged types.
* Update type validation to appropriately differentiate between type definitions and attribute templates.
* Update cache to appropriately handle different variations of attribute templates and type definitions.
* Update type manager client to support creating and editing of nested template groups.
* Add support for retrieving layout info for nested template groups.