Chapter 2: Bento Requirements

These requirements have emerged as a consensus from consideration of a wide range of usage scenarios, and from detailed discussions with developers who will use Bento. At this point they are quite stable, but we anticipate some further evolution as we understand better how Bento will be used.

Primary Objectives

These objectives define the basic reasons for developing Bento, and the essential goals that it must meet.

Content neutral

The container format must store any kind of content efficiently, and must not be biased toward any particular kind of content. The container itself should be clearly separated from the content, so that the same software can be used to access the container regardless of what kind of content it holds.

Platform neutral

The container format must work well on any platform. It must not be biased toward any particular hardware or system software environment.

Concrete format

The container format must fully define the concrete details of the container (where the bits go), not **just** an API. The specifications must be clear and complete enough to ensure that developers who implement the format can interoperate.

Good for random access read

The container format must allow efficient access to individual objects, given their IDs, when a container is stored on a random access device (memory, disk, CD-ROM, etc.).

Allow for update in place

The format must allow for implementation of full storage reuse, so that the Bento format can be used as a native document format in a full range of applications.

Derived Requirements

Given the primary objectives above, we derived the following requirements, largely by considering usage scenarios from many different domains: multimedia, compound documents, cross-platform transport standards, application data interchange, and others. Each requirement below is supported by several usage scenarios.

Simple and small

The container format needs to be simple, to make it quick to implement, flexible, and "light weight" in terms of memory and processor support.

<u>Standard API as well as format</u>

Developers must be urged to access the format through an API, rather than creating their own reader/writer code from the format spec. This will make the format much more evolvable and will speed adoption.

The API must be able to support update in place usage, as well as write-once, to allow for a graceful transition to enhanced functionality when it becomes available.

<u>External reference mechanism</u>

The format must support external references in and out of the container. These references must be rebindable when a container is moved, and the mechanism for storing references must be extensible over time so that new reference semantics can be added.

<u>Persistent references</u>

Objects must have persistent identity so that they can be reliably referenced across many versions of a given file. If some objects are deleted and others are added, the old references should either (1) continue to successfully refer to the correct object, or (2) detectably fail to refer to any object.

<u>Local generation of globally unique names</u>

Names of externally meaningful entities (such as types, properties, methods, etc.) must be globally defined across all Bento containers, so that software that depends on understanding the entities can recognize them. However, at the same time, not all such names can be registered, because types, properties, methods, etc. will be developed by very large numbers of “casual” developers, such as spreadsheet and stackware programmers, in addition to “professional” developers who could register their work.

Thus, Bento must provide a mechanism for naming types, properties, etc. that can be used by large numbers of developers without registration, and still provide reliably unique names. Registration must still be supported for common names.

<u>Object extensibility</u>

Objects must be extensible. Applications must be able to attach new information to any object without disrupting other applications that don’t understand the new information. Any information attached must have its own type, and must be able to be as large as any other part of the object. These requirements allow inspectable information to be associated with objects that would otherwise be opaque in some environments.

<u>Support for Recursive Access and Embedded Streams</u>

Values may themselves be containers or streams. Recursive access to embedded containers (either Bento or other formats) must be convenient and must support existing container usage. Access to streams must be convenient and must support existing stream usage.

<u>Encoding defined per value</u>

Objects created on different platforms, using different encoding standards, must be able to coexist within a standard container. This implies that byte-ordering, etc. must be defined at the value format level.

Multiple data formats for a single object

Sometimes an object used in multiple environments will need to be stored in multiple formats. The Mac clipboard and edition files are current examples. OLE is also an example: an embedded object is typically stored in two or three formats. In a multi-platform situation, in which a file is on a file server, being accessed from different environments, even more extreme usage scenarios along this line are likely to emerge.

Consistency checking

Applications using a container must have some way to tell which objects and values were updated when. This is important so that when we have multiple formats, related properties, etc. an application can at least detect which ones are consistent, and which ones may be inconsistent.

Maximum possible layout flexibility

The container format should impose the least possible constraint on the structure and location of values, allowing them to be stored in any order, broken up into pieces and interleaved, even nested inside of other values. This is necessary so that any bizarre requirements on layout can be met, including interleaving required for real-time playing of multi-media, constraints imposed by other standards, etc.

Out-of-line metadata

The container format must support values with no in-line type, size etc. This is necessary so that the format bytes are not in the way during reading some stream formats (including interleaved multi-media).

Inspectable references from each object

An application must be able to identify all objects that are referenced by a given object, **without** knowing the details of the value formats in the first object. Such a mechanism is required to enable applications or utilities to copy object structures,, delete object structures without dangling references, etc. without completely understanding all the formats of all the objects.

Remappable references from each object

An application must be able to change the target of references in a given object, without knowing the details of the value formats. Such a mechanism allows an application or utility to fix up the references when copying an object structure. This fixup is required to make the references point correctly to the copied objects, rather than the original objects. It is also useful in a variety of other contexts.