EV Object Types (Draft)

Following the ODP model, there are four types of Engineering Viewpoint objects: **Engineering Objects** (EV Objects) **Basic Engineering Objects** (EV BEO), **Container Objects** (EV Container) and **Channel Objects** (EV Channel). EV Objects provide functionalities which enable BEO distribution, EV BEOs map one to one with CV objects from the CV, EV Containers are used to group EV Objects and EV BEOs, and EV Channel objects are used to connect EV BEOs across containers [1].

Engineering Objects

An engineering object (EV object) is any object of interest in this viewpoint. For this reason BEOs, **Container Objects**and **Channel Objects**are also engineering objects (specialisations of). EV objects support computational requirements, distribution transparencies or infrastructure aspects of the system. The main distinction is made between BEOs, which represent computational objects (**Computational Viewpoint**), and other engineering objects, whose aim is to provide basic engineering functions, such as managers, interceptors and directories [2]

In the ENVRI RM, **Computational Viewpoint** are generic and can be used by any subsystem. In the EV, **Computational Viewpoint** can be specialisations to better describe a fully working subsystem. In practice, RIs may already have defined their systems and the division between subsystems may not be obvious (or as strictly defined as in the RM). In this case the subsystems will be viewed as logical not physical grouping of objects. This decomposition (logical or physical) can be used to identify practical interfaces for inter-operating between subsystems in different RIs. The different types of EV Containers allow the definition of both physically distributed and logically distributed containers. Thus, subsystems can be tightly coupled or extremely loose. The definition of EV channels will depend on the distribution model selected.

Basic Engineering Objects

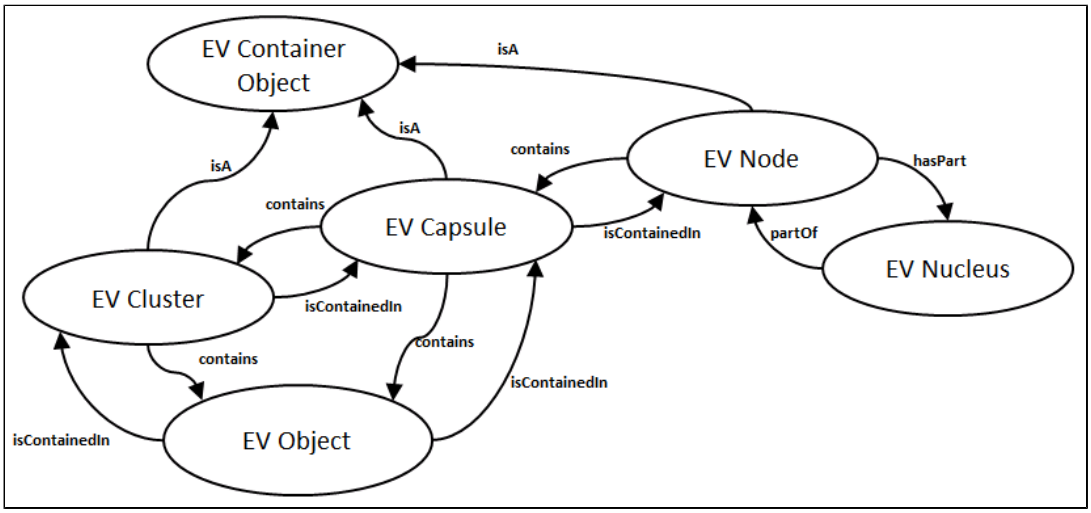
A Basic Engineering Object (BEO) is a special kind of **Engineering Objects**, which is used to give a representation of a computational object in the engineering viewpoint. The EV is concerned primarily with the engineering of machines and of network communications. Some computational objects may represent human actors, but for these there is just a simple placeholder BEO; the engineering of communication with them is a matter for HCI standardisation, but is not detailed in the reference model and so it is not discussed further here [3].

The set of BEOs can be seen as abstractions of the computational design. The resulting description hides distinctions between objects with similar communications requirements, and retaining only the information about the computational objects that characterizes them as users of the distribution platform being provided. Therefore, the BEO is the primary object to be placed on a particular node, and which initiates communication across the network. All other engineering objects are secondary elements defined in an **Architectural Model**, whose goal is to provide the functions necessary to support distribution. This includes a variety of supporting objects, like repositories or directories, which are drawn from a set of common functions called ODP functions Footnote03.

For instance, if a component is exposed, it becomes a BEO, but if it is kept private then it is not part of the EV.

Container Objects

The engineering viewpoint language defines four types of container objects to describe the principal controlling elements involved in any engineering specification: node, nucleus, capsule and cluster [4]. The figure to the right shows the containment relationships between containers, and engineering objects.



A **node** represents a physical object that has computing, communication and storage capabilities, and generally has connections to other nodes. A node will, therefore, have one or more network addresses, and the elements deployed on the node can become network accessible. A laptop, a PC, and server machines are all nodes. However, this is a slight simplification because a node does not actually have to be a separate, physically tangible object; a virtualization of such a resource can also be considered to be a node (a virtual machine for instance) Footnote04.

A **nucleus** represents the basic mechanisms needed to make a node function at the lowest level, typically representing an operating system kernel that manages and allocates processing capabilities, communication capabilities and storage capabilities. Fair scheduling and accurate timing both depend on the centralised control of the node's resources offered by the nucleus Footnote04.

A **capsule** is a unit of independent processing and storage. Faults within a capsule can affect all of the objects in it. The capsule supports a collection of engineering objects managed by a capsule manager. Capsules are isolated from one another by some protection mechanism, so that incorrect behaviour in one capsule does not damage other capsules. One of the consequences of this is typically that, because of the extra checking involved, communication across capsule boundaries is much more expensive than communication within a capsule. An example of a set of capsules is the set of independent processes, each with its own address space, run by an operating system (a nucleus). Another example of a capsule is a JEE or CCM component container Footnote04.

A **cluster** is a collection of **Basic Engineering Objects** that have closely coupled lifecycles, and so can be activated, deactivated or migrated as one single unit. The record of objects that makes up a JEE or .NET component is a cluster. Another example is the aggregation of small primitive objects into a larger configuration to form a row for database update [Footnote04].

Channel Objects

The engineering viewpoint is concerned with defining a channel architecture that represents the communication infrastructure, which allows engineering objects to interact. The basic element is the channel, which is the engineering equivalent of a computational binding. A channel consists of stubs, binders, protocol objects and interceptors, communicating **Basic Engineering Objects**, generally residing in different nodes [5].

Stubs transform or monitor information in the channel. This includes, for example, the marshalling and unmarshalling of message elements, the translation of local interfaces into interoperable interface references, or provision of message content encryption. Stubs are the elements that enable access transparency in the communication between two objects written in different languages (such as C++ and smalltalk). The client object talks to its local stub, which is in charge of translating the request into a neutral format that is sent along the channel and that the server stubs understand. The received request is then translated by the server stub to the language of the server, and passed to it. The response from the server follows a similar route back to the client, with the stubs again translating the messages. The result is that the client and server objects both think that they are talking to local objects written in their own language [Footnote05].

Binders provide services to establish a distributed binding between the BEOs communicating through the channel and to provide the transparency functions that coordinate replicated object instances. There can be a number of different dialogue styles involved in this. Thus, for instance, the client and server binders can set up the communication channel and the server binder can wait for requests before activating the server object. In fact, the server binder can exhibit different behaviours depending on the activation policy required for the server object; the binder can create one object for every request received, or instantiate only one object to take care of all incoming requests when the channel is started, or it can create one server object that takes care of all requests received during a period of time, but which terminates if it receives no requests for a while; many other instantiation policies are possible [Footnote05].

A **protocol** object is an encapsulation of the communication capability of the protocols, which may be a full stack of layered protocols for a specific task, such as support for the Web Services protocol SOAP. At a lower level, protocol objects might exploit IPv6 roaming support to keep in contact with mobile devices in a way that does not involve the recreation of bindings. A protocol object may also encapsulate an implementation of some special purpose protocol, such as a driver for a noise resistant satellite link or a quantum cryptography channel [Footnote05].

Interceptors are the elements provided by the engineering viewpoint to provide a gatewae when there is a need to cross organizational, security, system management, naming, or protocol domain boundaries. The fact that all the messages exchanged go through the channel interceptor also enables the addition of interesting management functionality to the system. For instance, messages can be observed in order to carry out quality of service and performance monitoring, or even reordered or filtered for security or other reasons. Another use is where two objects live in different networks, each following some local communication protocol; one uses OSI's seven layers architecture and the other one follows a vendor's proprietary protocol, for instance. The task of bridging these differences can be carried out by an interceptor within the engineering channel that connects them. Interceptors need not analyse all the layers of encapsulation in the communication; they can just pass on information unchanged if it is already understandable to both sides [Footnote05].

The engineering viewpoint defines a referece architecture. This Architecture is defined in three parts: the component model, and the way the components are integrated. The integration of components is defined as the node architecture and the channel architecture. The

---

[1]See D5.2 p. 48. [**42**]

[2]See Linington et.al. p94 [**37**]

[3]See Linington et.al. p. 91-92 [**37**]

[4] See Linington et.al. p. 94-95 [**37**]

[5] See Linington et.al. p. 96-98 [**37**]