Module name will contain some of what Java package names do today.

ambiguousname.domainname

e.g.

crm.salesforce.com

reports.crm.salesforce.com (*or* crmreports.salesforce.com)

For reusability, any existing module can be imported into another module that is being developed. The goal of importing a module is to *virtually merge* the contents of the module being imported into the module being developed in as seamless of a manner as is possible. A compiled module can *reference* other modules that it imported, meaning that only references to the imported module will exist in the importing module, and that the imported module will have to be located and loaded and “linked” into the importing module as part of the loading process of the importing module. While it is expected that most modules will be *pulled in* by reference at runtime, there will also be an option to *pull in* other modules in their entirety at *build*-time, physically including them within the build artifact (i.e. within the containing module’s XVM file structure itself).

To accomplish this, modules can import modules. Consider the most obvious syntax that first comes to mind:

module app.myco.com {

import crm.salesforce.com;

}

There are three issues with the above syntax. First, nothing differentiates the module import syntax from the class import syntax. Second, it can lead to all-too-easy namespace collisions, because the namespaces of the including module and the included module would naturally overlap. Third, it does not differentiate between build-time inclusion and runtime-resolved linkage. To address these, the syntax is as follows:

// the following form creates a runtime dependency on an external module named

// crm.salesforce.com and “mounts” the entire crm.salesforce.com module such that

// its contents are all available under the name “sfdc”, such that the “util” package

// within crm.salesforce.com would be referenceable as “sfdc.util” from anywhere

// within this module

import module crm.salesforce.com as sfdc;

// the following has the same effect, except that at build time, it will physically

// *contain* (*include*, *nest*, *copy*) all the contents of the crm.salesforce.com module

// into this module at the location “sfdc”

import module crm.salesforce.com as module sfdc;

A module could theoretically have a default import location:

module crm.salesforce.com default sfdc {…}

Then it could be imported using the simpler form:

import module crm.salesforce.com;

Within a module, there is no root package; each package that occurs directly within a module must have a name (like the “util” example previously). Any functionality that would be provided by a root package (if one could theoretically exist) must instead be placed directly onto the module itself. So, in a sense, “module is the root”. In fact, from the perspective of the contents of the module, the module is indeed the root (and there is no visibility beyond that root). This does imply that a module has to be able to do everything that a package can do.

Regarding the language itself, it is illegal to import the language module by build-time inclusion, and it is always implicitly imported by reference as if the following import statement were used in each and every module:

import module ecstasy.xtclang.org as xtc;

Furthermore, the classes located in the root of the ecstasy module are also implicitly imported at any location that classes can be imported:

import class xtc.\*; // note that “class” is implied, and thus could be omitted

There are four different uses in which modules appear within an XVM file structure:

1. The file structure can represent the contents of an entire module;
2. The file structure can contains references to the contents of other modules;
3. The file structure can embed another module;
4. (@deprecated) The file structure can contain overlays of another module.

TODO: discuss how this module’s version(s) can be indicated

TODO: discuss how a dependency upon and preferences for specific version(s) can be indicated

--

A module contains packages and/or classes.

* The qualified name contains an unqualified name, optionally followed by: a ‘.’ followed by the domain name

A package contains packages and/or classes.

* A package may be an alias to an imported module
* A package is contained by a module or another package

Containment:

* Module can be contained by: n/a
* Package can be contained by: Module, Package
* Class can be contained by: Module, Package, Class, Method
* Method can be contained by: Module, Package, Class, Method, Property
* Property can be contained by: Module, Package, Class, Method (note that properties of Module, Package, and Class must be immutable, while properties of Class instances and Method are scoped by instance)

Question: How can a property be placed on e.g. a module, package, or class? (not an object – which is an instance of a particular class – but the class itself, i.e. “static”)

Question: Do properties on modules, packages and classes need to be immutable?

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Module | Package | Class | Method | Property |
| Module |  |  |  |  |  |
| Package | x | x |  |  |  |
| Class | x | x | x | x |  |
| Property | x | x | x | x |  |
| Method | x | x | x | x | x |