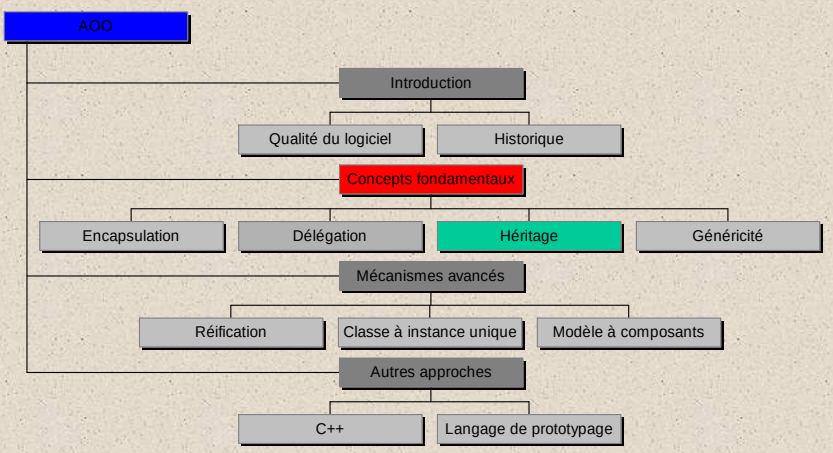
L'HERITAGE

TAXONOMIE DE L'HERITAGE

Sommaire



Sommaire

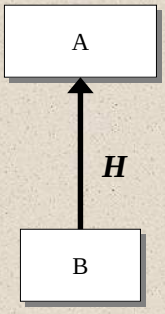
Taxonomie de l'héritage

Cela consiste à spécialiser la relation d'héritage.

- Cette spécialisation n'est pas nécessairement absolue
- Elle doit permettre une meilleure compréhension de l'utilisation de l'héritage.

Exemple de types particuliers d'héritage

- Spécialisation
- Réalisation
- Implantation
- Comportement
- Fusion
- Adaptation
- *Vue*



La classe B hérite de la classe A par un héritage de type *H*. La spécification de H se fera en indiquant
la définition informelle
les contraintes spécifiques

Spécialisation

Définition :

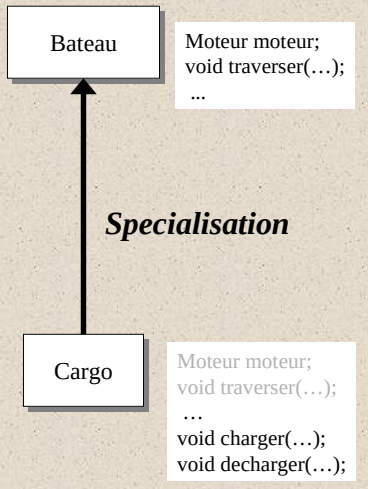
– une classe A est spécialisée par une classe fille B, si B ajoute des caractéristiques à la spécification de A

– Le concept couvert par A est raffiné par B

Contraintes :

– les caractéristiques exportées de A sont aussi exportées par B (B est un sous-concept de A). B peut exporter ses propres caractéristiques

• EXPORT(A) ⊆ EXPORT(B)



Réalisation

Définition :

– A est réalisée par B, si B décrit une implantation particulière de A

Contraintes :

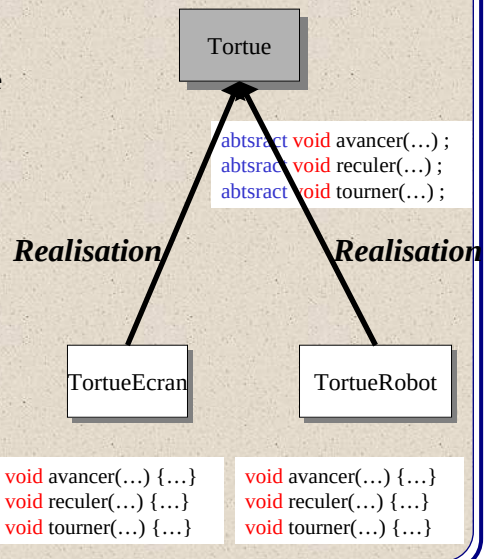
– A doit être une classe abstraite et B une classe concrète

• ABSTRAITE(A)

• not ABSTRAITE(B)

– les interfaces de B et de A sont identiques

• EXPORT(B) = EXPORT(A)



Implantation

Définition :

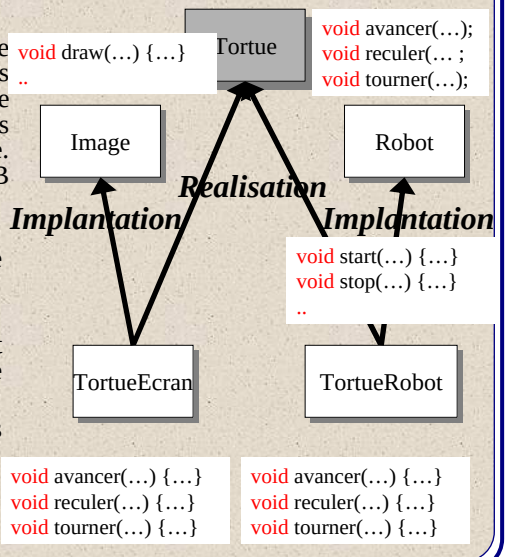
– une classe B est implantée par une classe A (A implante B), si les caractéristiques héritées de A ne sont utilisées qu'à des fins d'implantation de manière interne. Il existe une classe C telle que B hérite de C par Réalisation

Contraintes :

– les caractéristiques de A ne peuvent pas être exportées par B
  • EXPORT(A) ⊄ EXPORT(B)

– Les caractéristiques de A sont utilisées pour implanter celles de B

– A et B sont des classes concrètes C est une classe virtuelle.
  • Not ABSTRAITE(A),
  • Not ABSTRAITE(B),
  • ABSTRAITE(C)



Comportement

Définition :

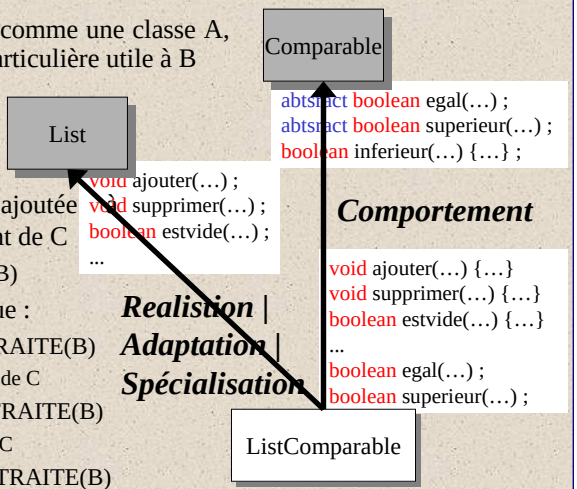
– Une classe B se comporte comme une classe A, si A décrit une propriété particulière utile à B

Contraintes :

– A doit être abstraite
  • ABSTRAITE(A)

– l'interface de A est ajoutée l'interface de B qui provient de C
  • EXPORT(A) ⊂ EXPORT(B)

– Il existe une classe C tel que :
  • ABSTRAITE(C) & ABSTRAITE(B)
    – B hérite par spécialisation de C
  • ABSTRAITE(C) & !ABSTRAITE(B)
    – B hérite par réalisation de C
  • !ABSTRAITE(C) & !ABSTRAITE(B)
    – B hérite par adaptation de C



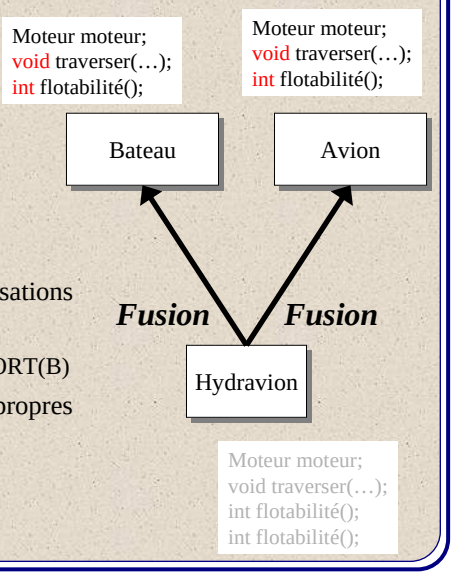
Fusion

Définition :

– une classe B fusionne deux classes A1 et A2 si les deux liens peuvent être vus comme des liens de spécialisation indépendants.

Contraintes :

– Les contraintes de chacune des spécialisations s'ajoutent
  • EXPORT(A1) ∪ EXPORT(A2) ⊆ EXPORT(B)
– B n'a pas de caractéristiques propres exportées.



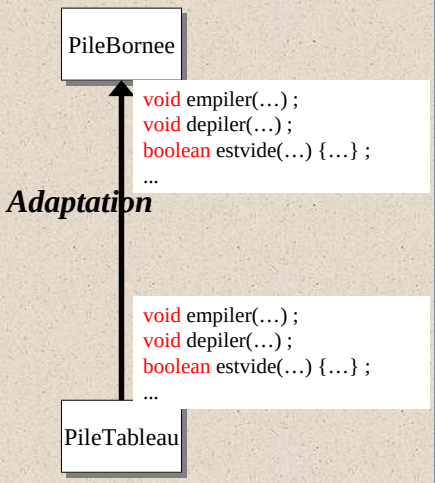
Adaptation

Définition :

– Une classe B adapte une classe A si B raffine sans ajouter de nouvelles caractéristiques.

Contraintes :

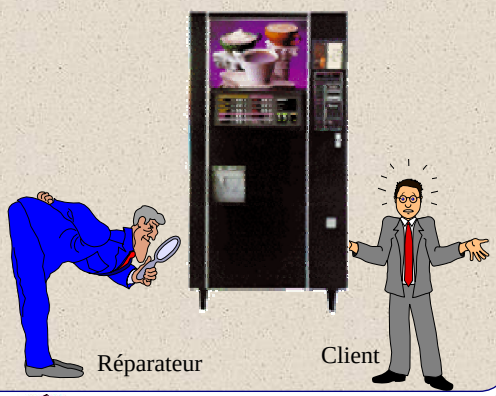
– A et B peuvent être abstraites ou concrètes, mais de manière simultanée.
– Aucune caractéristique n'est ajoutée
– B redéfinit certaines caractéristiques ou ajoute éventuellement de nouvelles contraintes invariantes.



Vue

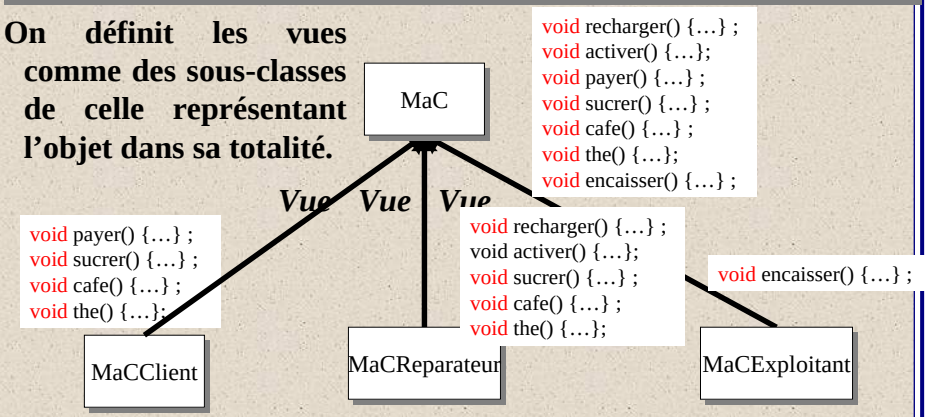
Une vue est une restriction des compétences d'un objet destinée à un type particulier d'utilisateurs

- Permet de fixer les compétences accessibles en fonction du type de l'utilisateur
- Les vues peuvent avoir des intersections non vides : une compétence peut être accessible à plusieurs types d'utilisateurs.
- La vue n'influe pas sur l'intégrité de l'objet, elle n'offre qu'une interface dédiée.



Vue : approche incorrecte

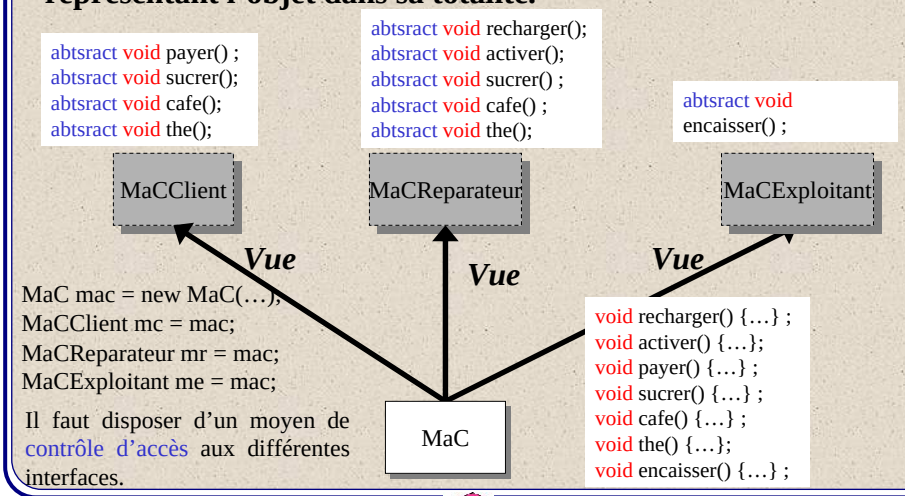
On définit les vues comme des sous-classes de celle représentant l'objet dans sa totalité.



- Impossibilité (en Java tout du moins) de restreindre l'export de la classe MaC.
- L'héritage impose la création d'objets de types MaCClient, MaCReparateur et MaCExploitant, alors que l'on ne veut qu'un seul objet.
- Le polymorphisme ne permet pas le mixage des 3 types.

Vue : approche comportement

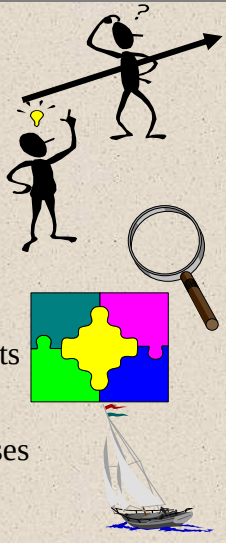
On définit les vues comme des abstractions de la classe représentant l'objet dans sa totalité.



Intérêts de la classification de l'héritage

Classifier l'héritage pour:

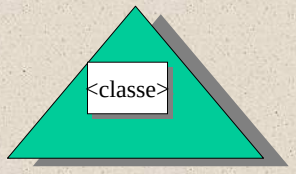
- apporter une sémantique aux liens d'héritage
- capturer les intentions du concepteur
- améliorer la lisibilité et la flexibilité
- fournir la base des nouveaux concepts structurants
- améliorer la navigation dans l'ensemble des classes
- *La classification donnée ici n'est pas universelle*



Structuration des classes

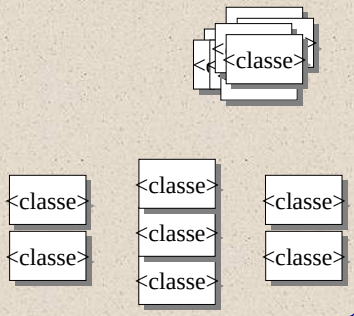
L'approche orientée objet conduit à :

- construire des unités de petite taille
- créer des variantes diverses de ces unités
  - généralisation
  - abstraction
  - …



D'où une prolifération importante des classes :

- difficile à gérer
  - conflit de nom
- difficile à exploiter
  - identification



Obligation de structurer cet ensemble de classes

Structuration

Structuration sans sémantique

- Cluster
  - Regroupement arbitraire de classes dans une unité
  - La sémantique de l 'unité est limitée
    - Container (Cluster)
  - Possibilité d 'organisation hiérarchique

Structuration avec une sémantique

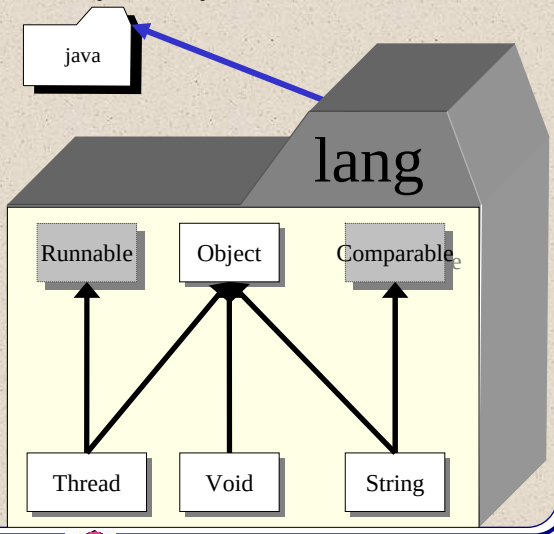
- Existence de concepts structurants
  - Sémantique imposant des contraintes
    - La nature des éléments
    - La nature des relations entre ces éléments

Package Java

Le package est à Java ce que le directory est au système de fichiers !

- il est une unité structurante
- il est une unité de portée
- structure l'espace des noms

- public class Visible { …}
- class Locale {…}
- public class Visible {
  - public int publique;
  - private int private;
  - protected int protected;
  - /*package*/ int packaged;



Void

```
/*
 * @(#)Void.java 1.6 98/09/21
 * Copyright 1996-1998 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */

package java.lang;

/**
 * The Void class is an uninstantiable placeholder class to hold a
 * reference to the Class object representing the primitive Java type void.
 * @author  unascribed
 * @version 1.6, 09/21/98
 * @since   JDK1.1
 */
public final class Void {
    /**
     * The Class object representing the primitive Java type void.
     */
    public static final Class TYPE = Class.getPrimitiveClass("void");

    /*
     * The Void class cannot be instantiated.
     */
    private Void() {}
}
```

La classe est accessible depuis l'extérieur du package

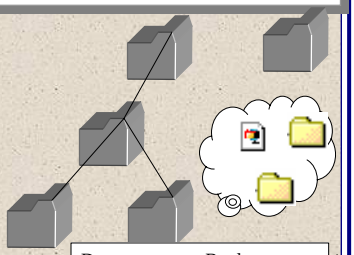
La classe ne peut avoir de descendance

Importation

Le désignation unique d'une classe se fait désormais par le chemin complet :

<Package>.<Package> … .<Class>

```
package essai;
class ENTIER {
    …
}
```



```
package essai;

import java.lang.Boolean;
import java.util.*;

public class INDICE extends ENTIER {
    …
    public java.lang.String  toString() {
        return '['+Binf+'/'+Valeur+'/'+Bsup+']'+new Date();
    }
    …
}
```

Dans cet espace Boolean est équivalent à java.lang.Boolean

Dans cet espace toutes les classes du package java.util sont accessibles par leurs noms relatifs.

Par défaut un nom relatif est complété par le package courant

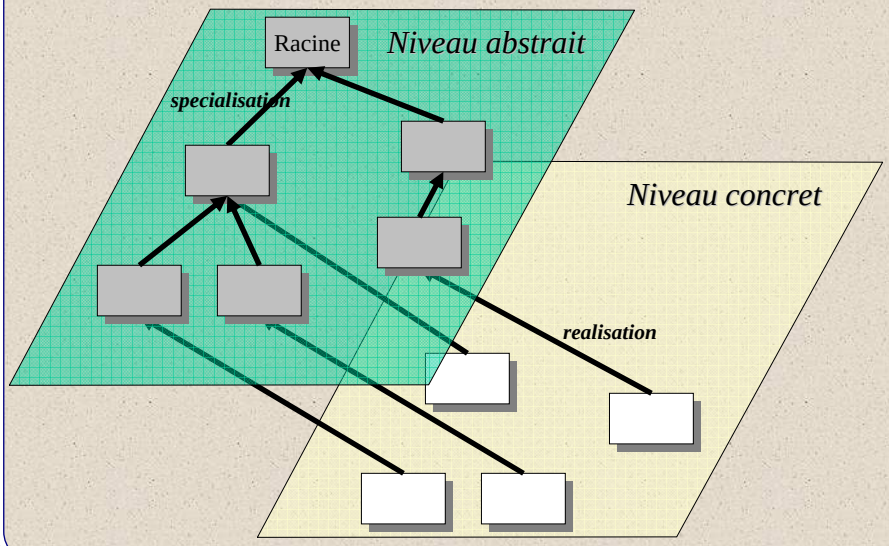
Désignation absolue de la classe

Sémantique de structuration : Domaine

Le Domaine est un ensemble de classes abstraites et concrètes qui sont nécessairement connectées par des liens forts d'héritage

- Règles de construction des Domaines
  - Une seule racine qui est abstraite (interface)
  - Les nœuds abstraits sont connectés par des héritages de spécialisation
  - Les nœuds concrets sont connectés aux nœuds abstraits par des héritages de Réalisation.
- Structure bi-niveaux
- Comment construire un Domaine
- Comment utiliser un Domaine

Domaine



Construction d'un Domaine

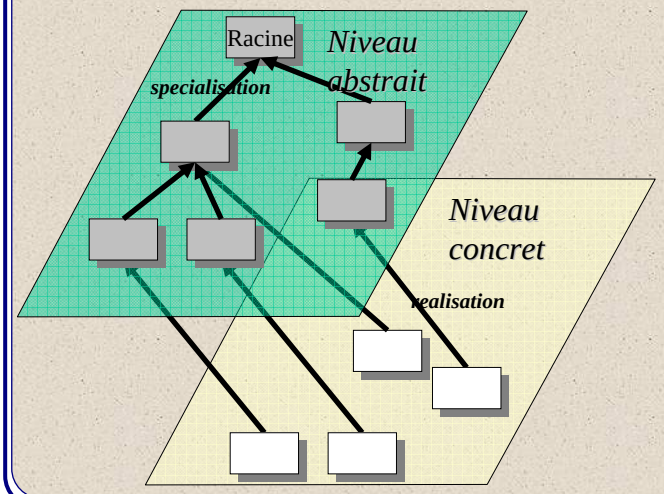
Les Domaines permettent de

- Capturer et raffiner les concepts majeurs d'une application ou d'un domaine d'applications
- Construire des composants réutilisables robustes
- Gérer le versionnement

Les domaines peuvent être construire indépendamment d'une application spécifique

- De toute pièce (from scratch)
- A partir d'un existant

Utilisation d'un Domaine



(relation de clientèle)

Client utilisateur

Client générateur

(relation de création)

Les domaines en Java

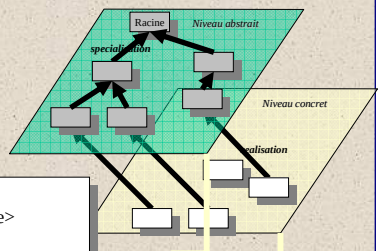
Pour représenter les domaines en Java, on peut utiliser les packages.

- On crée un package contenant le niveau abstrait
  - Interfaces ou classes abstraites
- On crée un sous-package de nom générique (par exemple : concret)
  - Classes concrètes

Accéder à une classes

- Comme utilisateur
  - Avec une désignation absolue
  - Avec une désignation relative
- Comme créateur

Remarque : la structure hiérarchique des packages n'est pas utilisable




```
…
… <domaine>.<classe>
…
```

```
import <domaine>.*;
…
… <classe>
…
```

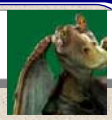
```
…
… <domaine>.concret.<classe>
…
```

```
import <domaine>.<concret>.*;
…
… <classe>
…
```

<domaine>

concret

JAR…JAR



general main attributes

- Manifest-Version:
  - Defines the manifest file version. The value is a legitimate version number, as described in the above spec.
- Created-By:
  - Defines the version and the vendor of the java implementation on top of which this manifest file is generated. This attribute is generated by the `jar` tool.
- Signature-Version:
  - Defines the signature version of the jar file. The value should be a valid *version-number* string.
- Class-Path:
  - The value of this attribute specifies the relative URLs of the extensions or libraries that this application or extension needs. URLs are separated by one or more spaces. The application or extension class loader uses the value of this attribute to construct its internal search path.

JAR…JAR

attribute defined for stand-alone applications This attribute is used by stand-alone applications that are bundled into executable jar files which can be invoked by the java runtime directly by running "`java -jar <x>.jar`".

- Main-Class :
  - The value of this attribute defines the relative path of the main application class which the launcher will load at startup time. The value must *not* have the `.class` extension appended to the class name.

JAR…JAR

attributes defined for applets These attributes is used by an applet which is bundled into JAR files to define requirements, version and location information for the extensions which this applet depends on. (see Extension Versioning **).**

–Extension-List:

•This attribute indicates the extensions that are needed by the applet. Each extension listed in this attribute will have a set of additional attributes that the applet uses to specify which version and vendor of the extension it requires.

–<extension>-Extension-Name :

•This attribute is the unique name of the extension. The Java Plug-in will compare the value of this attribute with the Extension-Name attribute in the manifests of installed extensions to determine if the extension is installed.

–<extension>-Specification-Version

•This attribute specifies the minimum extension specification version that is required by the applet. The Java Plug-in will compare the value of this attribute with the Specification-Version attribute of the installed extension to determine if the extension is up to date.

–<extension>-Implementation-Version

•This attritute specifies the minimum extension implementation version number that is required by the applet. The Java Plug-in will compare the value of this attribute with the Implementation-Version attribute of the installed extension to see if a more recent implementation needs to be downloaded.

–<extension>-Implementation-Vendor-Id

•This attribute can be used to identify the vendor of an extension implementation if the applet requires an implementation from a specific vendor. The Java Plug-in will compare the value of this attribute with the Implementation-Vendor-Id attribute of the installed extension.

–<extension>-Implementation-URL

•This attribute specifies a URL that can be used to obtain the most recent version of the extension if the required version is not already installed.

JAR…JAR

attributes defined for extension and package versioning **and** sealing **information These attributes define features of the extension which the JAR file is a part of. The value of these attributes apply to all the packages in the JAR file, but can be overridden by per-entry attributes.**

–Implementation-Title:

•The value is a string that defines the title of the extension implementation.

–Implementation-Version:

•The value is a string that defines the version of the extension implementation.

–Implementation-Vendor:

•The value is a string that defines the organization that maintains the extension implementation.

–Implementation-Vendor-Id:

•The value is a string id that uniquely defines the organization that maintains the extension implementation.

–Implementation-URL:

•This attribute defines the URL from which the extension implementation can be downloaded from.

–Specification-Title:

•The value is a string that defines the title of the extension specification.

–Specification-Version:

•The value is a string that defines the version of the extension specification.

–Specification-Vendor:

•The value is a string that defines the organization that maintains the extension specification.

JAR…JAR

attributes defined for file contents:

–Content-Type :

•This attribute can be used to specify the MIME type and subtype of data for a specific file entry in the JAR file. The value should be a string in the form of *type/subtype*. For example "image/bmp" is an image type with a subtype of bmp (representing bitmap). This would indicate the file entry as an image with the data stored as a bitmap. RFC 1521 and 1522 discuss and define the MIME types definition.

attributes defined for package versioning and sealing information

These are the same set of attributes defined above as main attributes that defines the extension package versioning and sealing information. When used as per-entry attributes, these attributes overwrites the main attributes but only apply to the individual file specified by the manifest entry.

attribute defined for beans objects:

–Java-Bean:

•Defines whether the specific jar file entry is a Java Beans object or not. The value should be either "true" or "false", case is ignored.