



# **Dependency Wiring for Unbundled Software Policies and Best Practices**

Version 0.5  
September 14, 2007

DRAFT

# Table of contents

<b>1 INTRODUCTION.....</b>	<b>3</b>
1.1 DEFINITIONS.....	3
<b>2 GENERAL GUIDANCE.....</b>	<b>4</b>
2.1 MINIMIZE HARD DEPENDENCIES.....	4
2.2 LEVERAGE STABILITY OF \$COMPONENT_ID.....	4
2.3 USE RELATIVE PATHS.....	4
2.4 HANDLE CHANGE IN \$INSTALL_HOME.....	4
2.5 DYNAMICALLY COMPUTE INSTALL LOCATION.....	4
2.5.1 Bourne Shell.....	4
2.5.2 Windows BAT.....	5
2.5.3 Perl.....	6
2.5.4 Python.....	6
2.5.5 Ruby.....	6
2.5.6 C: Unix.....	6
2.5.7 C: Windows API.....	8
2.5.8 Java.....	8
<b>3 PACKAGES AND PACKAGE DEPENDENCIES.....</b>	<b>9</b>
<b>4 LOCATING EXECUTABLES, UTILITIES, DATA FILES.....</b>	<b>9</b>
<b>5 NATIVE LIBRARIES (SHARED OBJECTS).....</b>	<b>10</b>
5.1 CONSTRUCTING SHARED OBJECTS.....	10
5.1.1 Solaris.....	10
5.1.2 Linux.....	11
5.1.3 Windows.....	11
5.2 RESOLVING DEPENDENCIES ON NATIVE LIBRARIES FOR EXECUTABLES.....	12
5.2.1 Solaris and Linux.....	12
5.2.2 Windows.....	12
<b>6 JAR FILES.....</b>	<b>12</b>
6.1 JAR VERSIONING.....	12
6.2 JAR DEPENDENCIES.....	13
6.3 WEB CONTAINERS.....	14
6.4 WEB APPLICATIONS.....	14
<b>7 LOCATING EXTERNAL DEPENDENCIES.....</b>	<b>14</b>
7.1 GENERAL APPROACH.....	14
7.2 JAVA SE.....	15
7.3 UNBUNDLED SOFTWARE INSTALLED IN OTHER LOCATIONS.....	15
7.4 OPERATING SYSTEM SERVICES.....	16
<b>8 INTERFACES.....</b>	<b>16</b>
8.1 INTERFACES IMPORTED.....	16
8.2 INTERFACES EXPORTED.....	16
<b>9 REFERENCE.....</b>	<b>16</b>

# 1 Introduction

This document is a companion to the *File System Layout for Unbundled Software: Best Practices and Requirements* specification (WSARC/2006/239). That specification defines how files should be organized under an installation umbrella directory. This specification defines how software components within that installation directory should resolve dependencies on each other.

## 1.1 Definitions

**Component:** a software component that makes up part of a software consolidation. A component could be as large as an application server, or as small as a library.

**\$COMPONENT\_ID:** a unique ID (name) that a component uses to name its subdirectories under the Install Home installation directory. Component Ids are discussed in WSARC/2006/239.

**Hard Dependency:** a dependency where a component will not function correctly if the dependency is not satisfied. Also known as an absolute dependency or a mandatory dependency.

**Install Home:** a self-contained umbrella directory that contains the installation image for a software consolidation.

**\$INSTALL\_HOME:** a shorthand notation to refer to the Install Home umbrella directory

**Soft Dependency:** a dependency where a component will still function correctly even if the dependency is not satisfied, although possibly with reduced functionality. Also known as an optional dependency.

## 2 General Guidance

### 2.1 Minimize Hard Dependencies

The more hard dependencies a project has the more complex and brittle it becomes (all else being equal). Therefore where possible projects should minimize the number of hard dependencies they have -- either by eliminating the dependency entirely (use `java.util.logging` instead of `log4j`), or converting it to a soft dependency (check if JESMF is installed before attempting to register with it).

### 2.2 Leverage Stability of `COMPONENT_ID`

As stated in WSARC/2006/239 projects are to consider their `COMPONENT_IDs` to be a committed interface. Therefore teams may leverage component Ids in order to locate a dependency.

### 2.3 Use Relative Paths

The relative paths within an installation umbrella directory are likely to remain very stable over the lifespan of the umbrella directory. On the other hand, the apparent location of the umbrella directory itself may change over time, or by use case. For example it could be renamed, or it could vary based on mount point.

Therefore use relative paths when specifying the location of something in the installation umbrella directory. Do not use absolute paths unless you construct them dynamically at runtime. Said another way: any static configuration file, library, jar manifest, etc. should never contain an absolute path to another file in the installation umbrella. Techniques for locating dependencies external to the umbrella directory are discussed in section 7.

### 2.4 Handle Change in `INSTALL_HOME`

Do not bake the value of `INSTALL_HOME` into any paths at install/config time. Make sure your code is robust enough to continue working if the user does:

```
$mv $INSTALL_HOME new_install_home
```

### 2.5 Dynamically Compute Install Location

As stated in sections 2.3 and 2.4 components should not hard-code the value of `INSTALL_HOME` into any paths. Instead executables should dynamically compute their installed location at runtime so that they can locate other files relative to themselves. This section provides some tips on the best way to accomplish that.

In the following sections the variable `my_home` (`MY_HOME`, `myHome`) is set to the fully canonical path of the directory that contains the executable that contains the specified code. Applications should compute `my_home` every time they are run in order to correct for any possible change in the location of the installation directory. In general the cost of doing this is assumed to be insignificant to the overall cost of application startup.

#### 2.5.1 Bourne Shell

Handling symlinks correctly complicates the Bourne shell solution.

```
#!/bin/sh

# Resolve a symbolic link to the true file location
resolve_symlink () {
    file="$1"
    while [ -h "$file" ]; do
        ls=`ls -ld "$file"`
        link=`expr "$ls" : '^.*-> \(.*\) '$' 2>/dev/null`
        if expr "$link" : '^/' 2> /dev/null >/dev/null; then
            file="$link"
        else
            file=`dirname "$1"`${"/$link"
        fi
    done
    echo "$file"
}

# Take a relative path and make it absolute. Pwd -P will
# resolve any symlinks in the path
make_absolute () {
    save_pwd=`pwd`
    cd $1;
    full_path=`pwd -P`
    cd $save_pwd
    echo "$full_path"
}

cmd=`resolve_symlink $0`
my_home_relative=`/usr/bin/dirname $cmd`
my_home=`make_absolute $my_home_relative`
```

Out of all the code snippets in this section, the Bourne Shell code appears to be the most inefficient. Even so, performance should not be a concern in most cases. A quick test on a 1GHz USIIIi system showed that the Bourne Shell code took roughly 0.016 seconds to determine `my_home` with one symlink in the path.

## 2.5.2 Windows BAT

For Windows BAT files the syntax, while cryptic, is much simpler. This will resolve to the true location of the `bat` file even if it is executed via a shortcut.

```
set MY_HOME=%~dp0
```

For the curious:

- ~ Starts a batch parameter modifier
- d Expands to drive letter of path
- p Expands to a full path
- 0 Expands to the batch file name

### 2.5.3 Perl

Perl has the `FindBin` module which supports locating the directory that contains the script being executed:

```
#!/usr/bin/perl

use FindBin;

$my_home = $FindBin::RealBin;
print "$my_home\n"
```

### 2.5.4 Python

Python does not have a `FindBin` equivalent, so you use `realpath` and `dirname`:

```
#!/usr/sfw/bin/python

import os
import sys

my_path = os.path.realpath(sys.argv[0])
my_home = os.path.dirname(my_path)
```

### 2.5.5 Ruby

Similar to Python. Requires `Pathname` library for `realpath` support:

```
#!/usr/local/bin/ruby

require 'pathname'

my_path = Pathname.new($0).realpath()
my_home = my_path.dirname()
```

### 2.5.6 C: Unix

For the previous scripting languages we had the luxury of knowing that `argv[0]` would either be an absolute path, or a valid relative path. This is because `argv[0]` is the script file that was passed to the interpreter. For native binaries on Unix `argv[0]` may be relative to one of the directories in `PATH` – not to our current directory. Also, it's possible `argv[0]` may not be valid at all if the binary was executed by some other poorly coded process. Therefore we use more robust interfaces where possible. Unfortunately these interfaces are not standard across Unix which leads to a more complicated solution than for the scripting languages.

The follow code sample should support the major Unixes. For Solaris it uses `getexecname()` and for Linux it reads the `execname` from the `proc` filesystem. In all cases the fallback method is to use `argv[0]` – searching `PATH` if necessary.

```

#include <stdio.h>
#include <stdlib.h>
#include <libgen.h>
#include <unistd.h>
#include <string.h>
#include <sys/param.h>

#ifdef __linux__
#define PROC_SELF_EXE "/proc/self/exe"
#endif

/*
 * get_exec_name
 *
 * Get the full path to the executable file that initiated this
 * process.
 *
 * cmd: The command used to execute this process. Typically
 *      argv[0]
 * execname: Buffer to hold result should be at least
 *           MAXPATHLEN+1 char
 * long
 *
 * Returns:
 * 0 On success
 * -1 An error occurred. Errno will be set.
 */
int
get_exec_name(const char *cmd, char *execname)
{
    char pathbuf[MAXPATHLEN + MAXNAMELEN + 1];
    const char *ename = NULL;
    char *s = NULL;

    memset(pathbuf, '\0', sizeof(pathbuf));

    /* Solaris and Linux have fairly robust ways to get the
     * exec name, so we try those first
     */
#ifdef __sunos__
    /* Use getexecname() on Solaris */
    ename = getexecname();
#elif defined(__linux__)
    /* Use procfs on Linux */
    if (readlink(PROC_SELF_EXE, pathbuf, MAXPATHLEN) == -1) {
        ename = NULL;
    } else {
        ename = pathbuf;
    }
#endif

    /* If we're not on Solaris/Linux, or those attempts failed,
     * we can use argv[0] if it contains a slash (i.e. if the
     * executable was invoked directly and not via a PATH
     * search)
     */
    if (ename == NULL && cmd != NULL && *cmd != '\0') {
        if (strchr(cmd, '/') != NULL) {
            ename = cmd;
        }
    }

    /* If we still don't have an exec name then try searching
     * PATH for it
     */
    if (ename == NULL) {
#ifdef defined(__hpux__) || defined(__sunos__)
        /* HP/UX and Solaris have pathfind */
        ename = pathfind(getenv("PATH"), cmd, "x");
#else
        /* Other platforms we do manually. There are more
         * efficient ways not using strtok, but we choose to
         * keep it simple.
         */
        s = strtok(getenv("PATH"), ":");
        while (s != NULL) {
            strcpy(pathbuf, s);
            strcat(pathbuf, "/");
            strcat(pathbuf, cmd);
            if (access(pathbuf, X_OK) == 0) {
                /* Found it */
                ename = pathbuf;
                break;
            } else {
                /* Check next path element */
                s = strtok(NULL, ":");
            }
        }
#endif
    }

    if (ename == NULL) {
        perror("Could not get exec name");
        return -1;
    }

    /* Make sure path is canonical with no symlinks */
    if (realpath(ename, execname) == NULL) {
        perror(ename);
        return -1;
    }

    return 0;
}

main(int argc, char *argv[])
{
    char execname[MAXPATHLEN + 1];
    memset(execname, '\0', sizeof(execname));

    get_exec_name(argv[0], execname);

    char *my_home = dirname(execname);
    printf("my_home=%s\n", my_home);
}

```

## 2.5.7 C: Windows API

For C on Windows we use the `GetModuleFileName()` and `GetFullPathName()` functions.

```
#include <windows.h>

main (int argc, char *argv[])
{
    DWORD retval=0;
    TCHAR pathBuf[MAX_PATH + 1];
    TCHAR myHome[MAX_PATH + 1];
    TCHAR *lpPart = NULL;

    /* Gets the path to the binary currently executing */
    retval = GetModuleFileName(NULL, pathBuf, MAX_PATH);

    if (retval != 0) {
        /* Canonicalize the path and strip off file name */
        retval = GetFullPathName(pathBuf, MAX_PATH, myHome, &lpPart);
        if (retval != 0 && lpPart != NULL) {
            *lpPart = '\\0';
        }
    }
}
```

### 2.5.8 Java

The recommendation for Java applications is to have your wrapper (either native code or a script) determine the install location using one of the techniques described in section 2.5, and pass that value to the Java program using a system property. Alternatively for stand alone Java applications the following approach appears to work:

```
import java.net.URL;

public class GetExecDir {

    public static void main(String args[]) {

        URL codebase = null;

        /* Get the codebase for this class */
        try {
            /* Cheesy way to get the Class object without hardcoding
             * the class name.
             */
            Exception ce = new Exception();
            Class c = Class.forName(ce.getStackTrace()[0].getClassName());
            /* Get codebase */
            codebase = c.getProtectionDomain().getCodeSource().getLocation();
        } catch (Exception e) {
            System.err.println("Could not determine code base: " + e);
            System.exit(1);
        }

        String my_home = codebase.getPath();

        /* If class came from a Jar, strip off jar file name leaving just
         * the directory that contained the jar file
         */
        int n = my_home.lastIndexOf(".jar");
        if (n > 0) {
            n = my_home.lastIndexOf(System.getProperty("file.separator"));
            my_home = my_home.substring(0, n);
        }

        System.out.println(my_home);
    }
}
```

### 2.5.9 Web Applications

[XXX Need to address web applications. How should they interact with the file system? Similarly for Mbeans in Cacao]

## 3 Packages and Package Dependencies

Details concerning package dependencies are outside the scope of this document. In general component's package dependencies should be consistent with their technical dependencies. For example if Component A has a soft dependency on Component B, then their packages should also have a soft dependency (for example the `Recommends` or `Suggests` fields in `.deb` packages).

Projects should avoid using the package system to locate their dependencies if at all possible. Relying on the package system has historically been a fragile approach. There are often variations across different OS environments, package names can change, and other details like the default `BASEDIR` can change which results in breaking software that assumed the original values.

## 4 Locating Executables, Utilities, Data Files

When an executable needs to locate another executable or data file residing in the same

`$INSTALL_HOME`, it should first determine its own location using one of the techniques described in section 2.6, then append the relative path to the file it depends on.

For example, if the shell script `$INSTALL_HOME/racerx/bin/rxadmin` depends on `$INSTALL_HOME/speedracer/bin/srdeploy`, then `rxadmin` should do something like:

1. Determine `my_home` as described in section 2.6.1
2. Append `../../speedracer/bin/srdeploy` to it
3. Invoke `srdeploy` using the path created in step 2

## 5 Native Libraries (Shared Objects)

In order to robustly resolve dependencies on shared objects (native shared libraries) two things must happen:

1. The shared libraries must be properly constructed
2. Binaries that depend on the shared libraries must properly state their dependencies.

### 5.1 Constructing Shared Objects

For a native library to be well behaved when interacting with its dependencies it must have two important characteristics:

1. It must be versioned
2. It must be self contained with respect to its dependencies

By versioning a library you ensure that other binaries don't mistakenly pick up the wrong version of the library (via a wayward `LD_LIBRARY_PATH` entry for example). By making the library self contained you avoid exposing dependencies your library has that your user should not need to worry about. This results in a simpler, more robust system.

How you accomplish these two characteristics varies by OS.

#### 5.1.1 Solaris

##### 5.1.1.1: Versioning

Since shared objects are loaded at run time the loader must be able to distinguish incompatible versions of a library. This is controlled by a version number associated with the shared object's name which is embedded in the shared object. You increment the version number any time you make an incompatible change to the library's interface.

Therefore when creating a library you must:

1. Ensure the library has a version number in its file name. For example: `libfoo.so.2`
2. Record that name in the `SONAME` field of the shared object during link time. For example: `cc -G -h libfoo.so.2 ...` (This ensures that any binary that links with your library knows what version to look for at runtime.).
3. Create an unversioned symlink to the latest version of your library (so your customers use the latest version when they build their binaries), and ship earlier versions of your library as well so you maintain backwards compatibility. In our example we end up with something like:

```
libfoo.so -> ./libfoo.so.2
libfoo.so.1
libfoo.so.2
```

### 5.1.1.2: Specifying Dependencies

Users of a library should not be required to know about the additional dependencies the library may have (and, on the flip side, they should be able to easily discover those dependencies if they happen to have a need to know) . Therefore when linking your library you should explicitly specify those dependencies, and at runtime your library should be constructed so that it can find them without the user setting `LD_LIBRARY_PATH`.

1. At link time specify all libraries your shared object requires. When using Sun's compilers you can use the `'-z defs'` option to the linker to make sure you get this correct. For example:

```
cc -G -z defs -lc -lnss3 . . .
```

2. At link time also specify the runpath that the loader should use to locate the shared object dependencies. Use the `$ORIGIN` token so that you can specify paths relative to the installed location of your shared object. For example:

```
cc -G . . . -R'$ORIGIN/.' -R'$ORIGIN/../../security/lib'
```

There are various other details one should understand when constructing shared libraries. For more information read the *Solaris Linker and Libraries Guide*. Also running `'dump -Lv'` on a shared object is handy for seeing it's dependencies and runpath settings.

## 5.1.2 Linux

### 5.1.2.1: Versioning

Please see the Versioning section for Solaris (5.1.1.1). For `gcc` you use the `'-soname'` option to specify the `SONAME` field.

### 5.1.2.2: Specifying Dependencies

Please see the Specifying Dependencies section for Solaris (5.1.1.2). For `gcc` you use the `'-rpath'` option to specify the runpath, and you may also use `$ORIGIN`.

On Linux you can use `'objdump -p'` to inspect shared object dependency and runpath settings.

## 5.1.3 Windows

Windows has a long and sordid history with respect to dynamic-link libraries (DLLs), versioning and dependency resolution. This situation, known as DLL Hell, was slightly improved with Windows 2000 where support for isolated applications was introduced. Windows 2003 and XP further improved this with the introduction of Side-by-Side Assemblies.

Unfortunately, as far as this author can tell, Side-by-Side Assemblies end up introducing more complexity than they are worth – at least as far as meeting the needs of multi-install. Therefore we propose a very simple approach.

[XXX Some apparent shortcomings:

1. While assemblies support specifying what version of a dependent assembly you need, you can only say “exactly version X.Y.Z.P” -- you can't say “X.Y or newer”. For example, if my application states a dependency on `foobar.dll` version 1.2.0.1, it must find that version **exactly**. If a bugfix is made to `foobar.dll` and it is now version 1.2.0.2 then my application won't use it. To get around this I must also ship an application config file that says 1.2.0.2 can be used in place of 1.2.0.1. Ugh.

2. There is no apparent way to specify a search path in an assembly manifest. This looks to be supported by .NET manifests, but not generic Windows manifests.
3. A “shared” assembly mitigates some of this – but those must be installed in a central OS location and therefore do not readily support multi-install.

If an expert on Windows has a knowledgeable recommendation in this area I'd like to hear it.]

#### Specific Guidance:

1. Version incompatible DLL's by incorporating a version numbers (likely major) into the DLL name and always ship all versions to maintain backwards compatibility. For example:
 

```
nss2.dll
nss3.dll
```
2. For private dll's (i.e. not shared components) keep the dll's in the component's bin directory along with the .exe that uses them. This guarantees the correct dll is loaded since the first place Windows searches for a dll is in the directory from which an application is loaded. [XXX what about public dll's that are not shared components. I.e. dll's shipped with a component that are a public API. Can those go in the component's 'bin' directory too?]
3. For shared component dll's any application that uses them must set PATH appropriately. See next section 5.2.2 for details.

## 5.2 Resolving Dependencies On Native Libraries for Executables

### 5.2.1 Solaris and Linux

Executables that depend on shared objects should use the same techniques as described in section 5.1.1.2. Specifically they should use `-R` and the `$ORIGIN` token to specify runpaths that are relative to their installed location. Executables must never require that the user set, or unset `LD_LIBRARY_PATH` to locate their dependencies, and they must never include absolute paths to locations in the installation umbrella directory in their runpaths.

### 5.2.2 Windows

Executables that depend on dynamic libraries should be started via a bat wrapper script. This script should prepend the location of any dll dependencies to `PATH` before starting the .exe. For example:

```
set PATH="%MY_HOME%\..\..\mps\lib;%PATH%"
```

Where `%MY_HOME%` is determined as described in section 2.6.2. One major drawback to this approach is if the dll exists in the system or Windows directory, then that version of the dll will be loaded, and not the one in `$INSTALL_HOME`.

Alternatively applications can uses explicit runtime loading [XXX need to expand]

## 6 Jar Files

In the case of dependency resolution, Jar files have many of the same characteristics as shared objects. Unfortunately Jar was firstly an archive format, and secondly a library/module format and therefore does not have mature mechanisms for versioning and dependency resolution. This will likely be solved by JSR-277, but for now this specification only deals with traditional Jar files (it is likely an addendum to this specification will be needed once JSR-277 hits the streets).

## 6.1 Jar Versioning

Public Jar files should be versioned, especially once they contain any incompatibilities. Versioning of Jar files is not as mature as that of shared objects, but a similar approach should be taken:

1. Set the `Specification-Version` and `Implementation-Version` fields in the Jar's manifest. The `Specification-Version` defines the version of the interfaces your Jar file supports, while the `Implementation-Version` specifies the version of the implementation of those interfaces. Examples:

```
Specification-Version: 4.2
Implementation-Version: 4.2.1.7
```

2. Add a version number to the jar file name once an incompatibility has been introduced. This version should be consistent with the `Specification-Version`. For example:

```
jss3.jar
jss4.jar
```

And ship all versions of the jar file to maintain backwards compatibility.

If your intent is to always maintain backwards compatibility then it is preferred to initially name the jar file without a version number (i.e. `jss.jar`) and only add a version number when incompatibilities are introduced.

In general you should only include the major version number in the jar file name. In exceptional cases where a minor number is also needed then it should be separated from the major number by an underscore: `jss4_5.jar`

3. Consider defining a `Main-Class` for your library jar files and have that class dump out your version and copyright information so the user can run “`java -jar yourjar.jar`” to easily get this information.

Some examples of jar incompatibilities are:

- A. Building the jar with a different version of the compiler such that the classes can't be used with an early version of the Java runtime.
- B. Modifying API's such that old code can no longer compile or run without changes
- C. ...[XXX]

## 6.2 Jar Dependencies

Jar files often have dependencies on other jar files – whether the jar file is acting as an application jar, or a library/module jar. The `Class-Path` field of the manifest should be set to resolve all class dependencies. The user of your application should not need to specify a `CLASSPATH` to get your application to run. Likewise the user of your library jar should not need to set `CLASSPATH` to resolve dependencies your jar file may have.

Specific guidance:

1. The jar file's manifest should specify the `Class-Path` field such that all dependencies for the jar file are satisfied.
2. The `Class-Path` entries must use relative paths for jar files residing in the installation umbrella directory. Note that the paths are relative to the physical location of the jar file itself – not the JVM's current directory, nor to any symlinks used to access the jar (this is all goodness).

For example the `Class-Path` field for `$INSTALL_HOME/myapp/lib/myapp.jar` might look like:

```
Class-Path: myutil.jar myl10n.jar ../../mps/lib/jss3.jar
```

3. If, for some reason, an application can't resolve all class file dependencies via jar file manifests, then it may set classpath (via one of the JVM mechanisms) in its wrapper script.

[XXX Need to address jar files in web containers. How does the Class-Path field work then?]

### 6.3 Web Containers

Web (and Java EE) containers are both consumers of jar files and providers of their interfaces. Historically web containers have used two approaches to resolve dependencies on these jars:

- A. Include the “shared component” version of the jar somewhere on their CLASSPATH. This has the benefit of leveraging a shared component, and picking up fixes to that shared component, but gives up control over the exact version of the API provided by the web container.
- B. Bundle a copy of the jar with the web container itself – often in the form of including the classes from the jar in the web container's “`rt.jar`” for improved class loading performance. The downside to this approach is the complexity of patching fixes for the jar. For example if `jaxm.jar` has a security fix multiple patches must be released. One for JAXM, plus one for each web container that ships a copy of JAXM.

With the advent of multi-install we now have the ability for consolidations to better control the version of shared components shipped in the consolidation with no worry of other consolidations swapping that version of the shared component out from under them. Therefore the main objection to approach A is reduced.

Specific Guidance:

1. Web containers should pick up shared component interfaces by including the shared component jar files on their classpath (approach A described above).
2. Where that is not possible then they may bundle copies of the jar files (or embed copies of the classes) in their own product. But this should be avoided if at all possible since it complicates deployment of fixes via patches/updates.

### 6.4 Web Applications

Specific guidance:

1. Where possible, web applications should use the interfaces provided by the web container itself. I.e. they should not bundle their own copy of jar files in their WAR files.
2. In those cases where they feel they must bundle their own copy of the interfaces they should construct their WAR files at deployment time – grabbing the shared component version of the interface.
3. Web applications must not bundle shared component interfaces (jars) at product build time.

[XXX This section overly simplifies a complex issue. Needs re-work]

## 7 Locating External Dependencies

In addition to dependencies that are resolved within the installation umbrella directory, it is likely that components will have dependencies on things that live outside of the umbrella directory. In this case the techniques described so far may not be appropriate.

## 7.1 General Approach

The general approach is to provide this information to components by using environment variables that are set via configuration files global to the installation umbrella directory. Specifically:

- Unix: `$INSTALL_HOME/etc/env.conf`
- Windows: `$INSTALL_HOME/etc/env.bat`

These files are created at installation time, with the assumption that the installer will populate them with initial values. The files can then be sourced by components at startup to gain access to the environment variables. Components can locate these files using the technique described in section 4.

Example content:

```
# /bin/sh
# env.conf
# Configuration parameters for Application Platform
# Format:
# property=value
# Do not use any other syntax in this file.
#
ENTSYS_JAVA_HOME=/usr/java
```

```
REM
REM env.bat
REM Configuration parameters for Application Platform
REM Format:
REM set property=value
REM Do not use any other syntax in this file.
REM
set ENTSYS_JAVA_HOME=C:\java\jdk1.5.0_09
```

To avoid namespace collisions the following conventions should be used when choosing variable names for use in these files:

1. Variables defined by the consolidation should be prefixed with “ENTSYS\_”
2. Variables defined by a specific component should:
  - A. Be an existing environment variable already declared as a public interface
  - B. Or be prefixed by the component's `$COMPONENT_ID` followed by an underscore (`_`)

Note that full Bourne shell and BAT syntax are supported in these files. This is because some values may need to be computed dynamically. The best example is Java SE on Windows where the BAT file may need to look the location of Java SE up in the Windows registry using `reg.exe`.

## 7.2 Java SE

The location of Java SE is specified by the `ENTSYS_JAVA_HOME` environment variable which is set using the mechanism described in section 7.1. This version of Java SE is assumed to be 1.5 or newer. If a component needs a specific version of the JVM they should use

'\$JAVA\_HOME/bin/java -version:<version>' to select a specific version (this capability was introduced in 1.5).

## 7.3 Unbundled Software Installed in Other Locations

Cross dependencies between unbundled software installed in disparate locations should be avoided as much as possible. But on occasion an application may need to access a command or utility that resides in another install location – for example if Portal Server wants to deploy into an existing Application Server deployment.

In this case the recommendation is to gather that location at configuration/deployment time via user input (aided by scanning of the system to locate potential locations if possible), and save the location as a full path, either in the global `env.conf/env.bat` file, or in private product configuration files. By using a full path we preserve the property of an installation continuing to function even if its umbrella directory is moved or renamed. On the other hand if the installation we depend on is moved we break – but that is unavoidable, and at least a natural consequence of moving a dependency.

## 7.4 Operating System Services

The general area of integrating with Operating System Services, such as SMF, Windows Services, etc. will be covered in a companion specification. But for simple OS libraries and commands the general rule is to use absolute paths to those libraries (when stating library dependencies) and commands (when specifying the location of the command). Do not use relative paths, and do not rely on the user's `PATH` environment variable to locate OS commands.

# 8 Interfaces

## 8.1 Interfaces Imported

INTERFACE	CLASSIFICATION	COMMENTS
Filesystem Layout for Multi-Install Architecture (WSARC/2006/239)	Committed	<a href="http://sac.sfbay.sun.com/Archives/CaseLog/arc/WSARC/2006/239/">http://sac.sfbay.sun.com/Archives/CaseLog/arc/WSARC/2006/239/</a>

## 8.2 Interfaces Exported

INTERFACE	CLASSIFICATION	COMMENTS
\$INSTALL_HOME/etc/env.conf	Committed	
\$INSTALL_HOME/etc/env.bat	Committed	
ENTSYS_JAVA_HOME environment variable	Committed	

# 9 Reference

[0]	Title: Multiple Installation and Management of Layered Products for Java Enterprise System (WSARC/2006/130) URL: <a href="http://sac.sfbay.sun.com/WSARC/2006/130/">http://sac.sfbay.sun.com/WSARC/2006/130/</a>
[1]	Title: Filesystem Layout for Multi-Install Architecture (WSARC/2006/239) URL: <a href="http://sac.sfbay.sun.com/Archives/CaseLog/arc/WSARC/2006/239/">http://sac.sfbay.sun.com/Archives/CaseLog/arc/WSARC/2006/239/</a>

[2]	Title: Solaris 10 Linker and Libraries Guide URL: <a href="http://docs.sun.com/app/docs/doc/817-1984">http://docs.sun.com/app/docs/doc/817-1984</a>
[3]	Title: Red Hat Enterprise Linux 4 Using ld, the Gnu Linker URL: <a href="http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/gnu-linker/">http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/gnu-linker/</a>
[4]	Title: Windows Dynamic-Link Library Functions URL: <a href="http://msdn2.microsoft.com/en-us/library/ms682599.aspx">http://msdn2.microsoft.com/en-us/library/ms682599.aspx</a>
[5]	Title: How To Build and Service Isolated Applications and Side-by-Side Assemblies for Windows XP URL: <a href="http://msdn2.microsoft.com/en-us/library/ms997620.aspx">http://msdn2.microsoft.com/en-us/library/ms997620.aspx</a>
[6]	Title: JAR File Specification URL: <a href="http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html">http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html</a>
[7]	Title: Requirements for Sun projects delivering on the Windows platform URL: <a href="http://sac.sfbay.sun.com/arc/WSARC/2002/494/opinion.txt">http://sac.sfbay.sun.com/arc/WSARC/2002/494/opinion.txt</a>
[8]	Title: RFC-2119: Key words for use in RFCs to Indicate Requirement Levels URL: <a href="http://rfc.net/rfc2119.html">http://rfc.net/rfc2119.html</a>
[9]	Title: Design Specifications and Guidelines - Integrating with the System URL: <a href="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwue/html/ch11b.asp">http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwue/html/ch11b.asp</a>
[10]	Title: Interface Taxonomy URL: <a href="http://sac.sfbay.sun.com/cgi-bin/bp.cgi?NAME=interface_taxonomy.bp">http://sac.sfbay.sun.com/cgi-bin/bp.cgi?NAME=interface_taxonomy.bp</a>
[11]	
[12]	